Trabajo de Fin de Grado

Grado en Ingeniería Informática (GEI)

Evaluación y optimización de algoritmos Fast Fourier Transform en SX-Aurora NEC

Facultat d'informática de Barcelona (FIB)

Barcelona Supercomputing Center (BSC)

Autor	:	Pablo Vizcaino Serrano
Director	:	Filippo Mantovani
Ponente	:	Jesus Labarta
Especialidad	:	Ingeniería de Computadores
Convocatoria	:	Junio 2020

Agradecimientos:

Quiero agradecer la ayuda de Constantino Gomez, compañero del BSC, por compartirme su experiencia con la máquina NEC SX-Aurora; y a Erich Focht, trabajador de NEC, por las consultas sobre mi implementación de la FFT que ha resuelto.

Índice

1. Introducción

Este proyecto es un Trabajo de Fin de Grado (TFG) del Grado de Ingeniería Informática (GEI) impartido por la Facultad de Informática de Barcelona (FIB). Este trabajo también se enmarca dentro del Barcelona Supercomputing Center(BSC).

Se estudiará el comportamiento de un conjunto de códigos que realizan la Transformada de Fourier en computadores de alto rendimiento.

Este trabajo presenta una optimización del cálculo de la Transformada de Fourier para arquitecturas con vectores largos. Al ser un cálculo muy presente en una gran variedad de códigos científicos, su optimización siempre es asunto de actualidad e investigación. En el documento encontramos un estudio exhaustivo del algoritmo, explorando diferentes posibles optimizaciones. Finalmente se expone un algoritmo para cálcular la Transformada de Fourier que se aprovecha de los beneficios de las arquitecturas vectoriales y consigue un mejor rendimiento que las librerías ya existentes en ciertos casos.

Normalmente las optimizaciones de los códigos en la computación de alto rendimiento se basan en el paral·lelismo, buscando aprovechar mejor la cantidad de CPUs y su comunicación. En canvio, en este trabajo nos centraremos más en optimizarlo para el hardware que tenemos. En concreto, optimizaremos el código para realizar un uso eficaz de un acelerador vectorial.

Este documento es el resultado de 5 meses de trabajo en el Barcelona Supercomputing Center (BSC), centro de computación de alto rendimiento que aloja el supercomputador Marenostrum, #20 en la Top500 (Nov 2019). El trabajo se ha llevado a cabo colaborando con la empresa NEC, que ha proporcionado información detallada sobre los componentes tecnológicos evaluados en este trabajo.

El trabajo está estructurado como sigue. En la sección 2 se definen todos los conceptos relacionados con el trabajo. En la sección 3 se explica la motivación del proyecto y el punto de partida. En la sección 5 se explica y evalúa un primer intento de optimización del cálculo de la Transformada de Fourier. En la sección 6 se muestra la implementación del algoritmo diseñado en este proyecto y se evalúa su rendimiento. Por último, en la sección 7 se recapitulan las conclusiones del trabajo.

2. Background

En esta sección se explican los conceptos previos relevantes para la buena comprensión del trabajo que se presentará en la parte restante de este documento. En primer lugar se introducirán los elementos hardware; el entorno del High Performance Computing (HPC) y las extensiones vectoriales. En segundo lugar se introducirá la función matemática en cuestión que se estudiará, las librerías matemáticas ya existentes que la implementan y los elementos fundamentales en el análisis de rendimiento.

2.1. High Performance Computing (HPC)

Este concepto se refiere a la práctica de agregar la potencia de cálculo de muchos computadores, consiguiendo así una capacidad de cálculo muy superior[1] al de los ordenadores de sobremesa o servidores. Comúnmente se utiliza esta potencia de cálculo para fines científicos, de ingeniería o de negocios.

A esta agregación de computadores también se le suele referir con el nombre de Supercomputador o Cluster. Existen muchos supercomputadores repartidos por diferentes países. Algunos de ellos están gestionados por grandes empresas privadas, y otros están al servicio de la comunidad científica a través de centros de cálculo o redes de excelencia de la computación como RES o PRACE. Estos últimos cuentan con la participación pública de estados y federaciones de estados. Los supercomputadores pueden diferir entre sí en función de qué arquitecturas utilizan, de quantos procesadores disponen, si utiliazan aceleradores, qué tipo de red de interconnexion tienen, etc...

Los componentes de un supercomputador se pueden observar de manera esquemática en la Figura 1

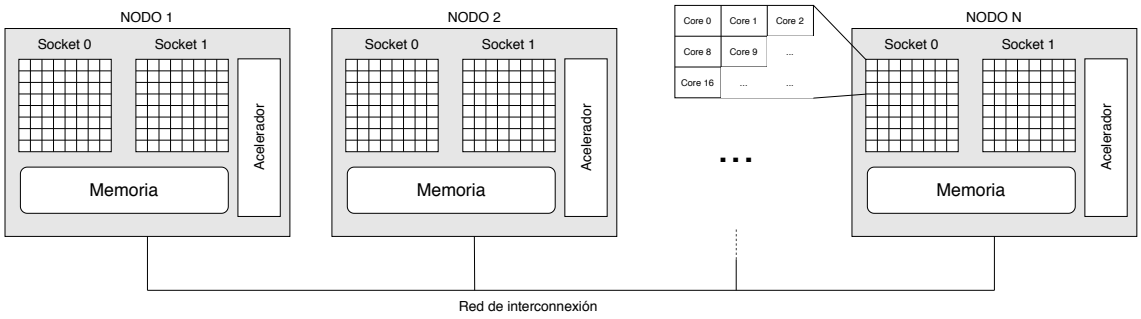

Figura 1: Diagrama de los componentes de un supercomputador.

Describiéndolo desde los elementos más internos hacia fuera, primero encontramos los cores o CPUs. Éstos se juntan en Sockets, que a su vez se integran en placas que pueden ser multisocket. Estas placas cuentan con memoria y, a veces, aceleradores como GPUs o unidades vectoriales. A este conjunto se le llama Nodo. Para comunicar los Nodos entre sí y formar el Supercomputador se utiliza una red de interconnexión, tal y como se observa de manera esquemática en la Figura 1.

El entorno HPC está en constante evolución. Aunque actualmente alguna arquitectura pueda ser más prominente, hay arquitecturas emergentes que intentan construirse un espacio en el mundo HPC. En esta tésis se evaluará alguna de estas arquitecturas emergentes y se valorará si requieren un cambio en la manera de programar respecto a las arquitecturas convencionales.

Este trabajo se centrará en el nivel bajo de la estructura de un clúster, midiendo el rendimiento a nivel de un solo core y de un acelerador.

2.2. Extensiones vectoriales

En esta tesis se utilizarán arquitecturas y máquinas muy variadas. Cada arquitectura puede ir acompañada de una extensión vectorial que permite realizar la misma operación a muchos datos a la vez. A estos cálculos vectoriales se les suele llamar SIMD[2] (Single Instruction Multiple Data). Estas operaciones son naturalmente útiles para el procesamiento de imágenes o vídeo, dado que se pueden utilizar en los filtros sobre las imagenes, realizando la misma operación sobre todos sus píxeles. Por otro lado, también sirven para realizar cálculos muy comunes en muchas aplicaciones, como por ejemplo la multiplicación de matrices y la multiplicación de vectores, entre otras.

Los compiladores actuales tienen la capacidad de autovectorizar nuestro códigos, pero a veces la estructura de códigos complejos no permite una autovectorización eficiente. Si el usuario quiere sacarle partido, puede usar los intrínsecos (llamadas concretas en el código) de su arquitectura. Cada arquitectura tiene un *set* de intrínsecos propio, hecho que complica la portabilidad de las apliaciones a otras máquinas con distintos intrínsecos.

En la Figura 3 se puede observar un ejemplo de vectorización utilizando intrínsecos. La multiplicación de dos arrays, que normalmente requeriría 2*N Loads, N Stores y N multiplicaciones se convierte en tan solo 4 instrucciones vectoriales: dos Loads, una multiplicación y un Store.

```
//A,B,C are double vectors
for(int i=0; i<N; ++i){
        C[i] = A[i] * B[i];
}
```

```
__vr v_A = _vel_vld_vssl(sizeof(double),A,N); //Load A
__vr v_B = _vel_vld_vssl(sizeof(double),A,N); //Load B
__vr v_C = _vel_vfmuld_vvvl(v_A,v_B,N); //C[i] = A[i]*B[i]
_vel_vst_vssl(v_C,sizeof(double),C,N); //Store C
```

Figura 2: Código original (izquierda) y vectorización con intrínsecos (derecha).

2.3. SX-Aurora TUBASA

Éste es el sistema de cálculo con el que se trabajará en este proyecto. el SX-Aurora[3] es un computador de NEC Corporation que está compuesto por una máquina x86, referida como *"Host"*, conectada por PCIe al *"Vector Engine"*. Éste último es un acelerador vectorial que cuenta con una Scalar Processing Unit (SPU) y una Vector Processing Unit (VPU) que permiten ejecutar de manera autónoma las aplicaciones. En el *Host* el usuario programa las aplicaciones y después de compilarlas con el compilador própio de NEC, pueden ejecutarse en el acelerador en vez de en el *Host*.

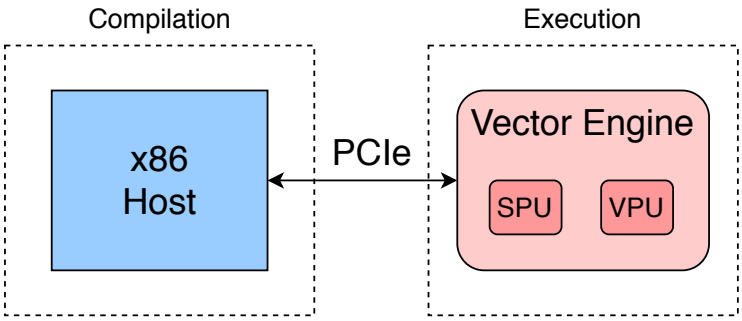

Figura 3: Diagrama del sistema Host - Vector Engine.

La unidad vectorial de NEC cuenta con registros de 256 elementos de tipos double (es decir, de 64 bits). Por lo tanto, llega a almacenar 16384 bits por vector. Este número está muy por encima del de las arquitecturas convencionales. Por ejemplo, Intel dispone de un máximo de 512 bits por registro con su extensión AVX-512[4]. En Arm tenemos la extensión Neon[5], de tan solo 128 bits y la extensión SVE[6] con registros de hasta 2048 bits.

En concreto, este proyecto se ha realizado en un nodo de esta arquitectura que dispone de un *Host* y dos *Vector Engines*. El hecho de disponer de dos aceleradores ha sido relevante, no por la possible paralelización sinó por poder trabajar concurrentemente con otros usuarios de la máquina, ya que ésta no dispone de un sistema de colas. Las colas balancearían la carga de trabajo entre los diferentes sistemas de cálculo y pueden asegurar la exclusividad de los recursos, de manera que no se produzcan conflictos indeseados entre dos procesos distintos.

El compilador de NEC recibe el nombre de *ncc*. Este compilador es similar al gcc, con la diferencia de que es capaz de auto-vectorizar el código para la arquitectura NEC. El problema del compilador *ncc* es que aún no dispone de instrucciones intrínsecas para poder vectorizar manualmente los códigos. Para realizar esta tarea, se utiliza un compilador *LLVM*, que a parte de dar soporte para la auto-vectorización, dispone de un largo conjunto de intrínsecos [7].

2.4. Transformada de Fourier

La Transformada de Fourier [8] es un algorítmo matemático que descompone una señal -por ejemplo de audio- en una suma de señales sinusoidales. Al hacer esta descomposición, pasamos de ver la señal en el dominio del tiempo al dominio frecuencial. Este proceso tiene diversas utilidades en muchos campos del procesamiento digital de señal.

El cálculo de la transformada de Fourier está en la base de muchos códigos de simulaciones científicas utilizados en High Performance Computing(HPC) y, por lo tanto, es un objeto de estudio muy relevante en la supercomputación. Una mejora muy pequeña en este código se convierte en una enorme al escalarlo a miles de procesadores.

Para entender mejor la transformada de Fourier, en las Figuras 4 y 5 se puede observar como aplicándola sobre una señal obtenemos las frecuencias que se han utilizado para crearla.

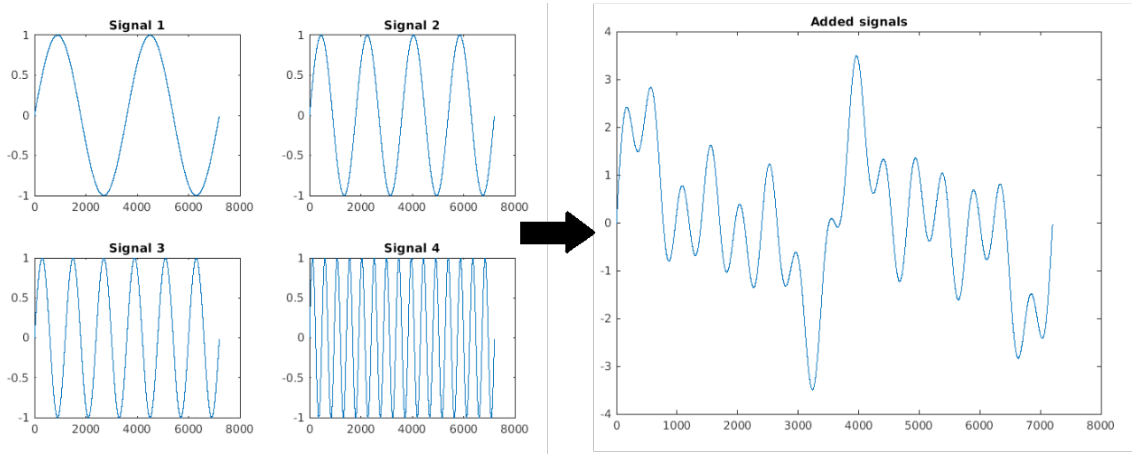

Figura 4: Creación de una señal a partir de la suma de cuatro señales sinusoidales.

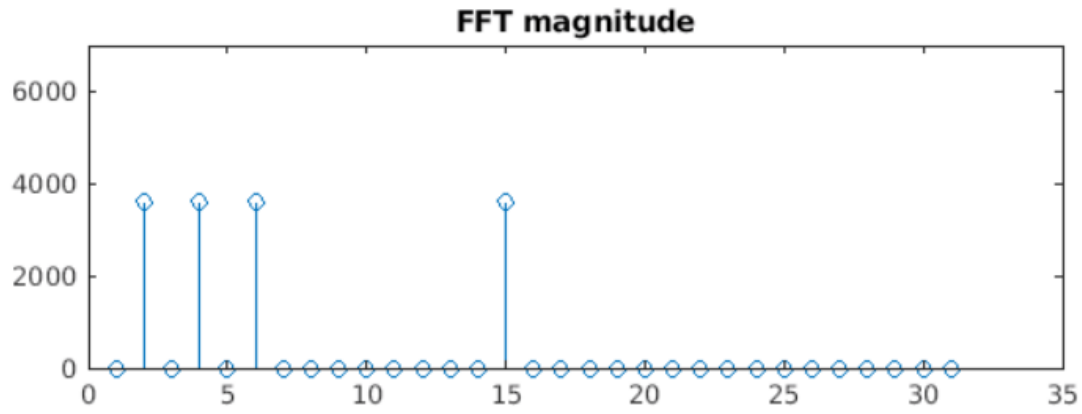

Figura 5: FFT de la señal agregada.

Observamos que hay un pico en la segunda, cuarta, sexta y quinceavaa posición. Estas posiciones corresponden con las señales sinusoidales utilizadas para crear la señal agregada. La primera tenía dos periodos en las 8000 muestras, la segunda cuatro, etc... El caso que se ha mostrado es una FFT ideal en que todas las señales tenían un número entero de periodos. De no ser así , la FFT pasa de tomar valores "binarios" como en la Figura (más grande que cero o igual que cero) a un abanico de valores aproximando el numero de periodos que tiene cada señal.

Uno de los usos de la transformada de Fourier se encuentra en el campo musical, en concreto para afinar la voz -el comúnmente conocido *Autotune* [9]-. La Transformada de Fourier permite obtener las notas musicales cantadas, que no dejan de ser frecuencias, aplicarles correcciones y realizar la Transformada de Fourier inversa

para volver a crear la señal completa.

También se puede usar la transformada en otros campos, como por ejemplo el alimentario. Muchas empresas venden aceite de oliva alegando su pureza pero no todas lo cumplen. Un estudio [10] utilizó la Transformada de Fourier sobre un conjunto de muestras de aceite de oliva para poder ver en el espectrómetro cuales eran sus componentes reales. Éste es un ejemplo más donde la Transformada de Fourier nos permite separar la información, que de otra manera sería clasificada como ruido, en pequeños bloques que podemos entender y manipular.

2.4.1. FFTW

Una de las implementaciones más comunes del algoritmo es la "Fast Fourier Transform" o, más comúnmente llamada por su diminutivo, la FFT[11]. Resumidamente, la FFT interpreta la Transformada de Fourier como un producto de matrizes (de allí el coste inicial N^2). Después utiliza propiedades de los números complejos para poder convertir una multiplicacion de dos matrizes en múltiples multiplicaciones más pequeñas y simples. Si se quiere saber más, en la referencia anterior se explica por qué esta implementación convierte el coste $N^2 \, en \, N \cdot log(N)$.

Una implementación más concreta de la FFT es la FFTW [12]. Esta implementación es una librería para los lenguajes de programación C y Fortran que tiene la peculiaridad de adaptarse a la arquitectura donde se ejecuta, cambiando automáticamente su configuración -por ejemplo, el orden de las operaciones-.

Aunque en esta introducción no se entrará en detalles técnicos -que se pueden consultar en la referencia anterior- FFTW utiliza un método de calibración previo donde compara diferentes versiones y opciones de sí misma en la arquitectura en que se está usando, de manera que puede escoger de manera automática la configuración óptima para un problema concreto. También está pensada para vectorizar, aunque no para las nuevas arquitecturas con vectores muy largos. En este proyecto se va a utilizar la FFTW como punto de referencia, ya que actualmente es una de las mejores -si no la mejor- implementaciones de la FFT.

La calibración previa que se realiza al utilizar la FFTW se denomina *plan*. Un *plan* puede ser ejecutado diversas veces con diferentes arrays de entrada y salida, mientras el tamaño de éstas se mantenga constante.

2.5. Librerías matemáticas

En los supercomputadores son muy habituales las librerías matemáticas específicas. Éstas implementan las operaciones matemáticas más comunes en los algoritmos de apliaciones científicas, como por ejemplo multiplicaciónes de matrizes, transposiciones y todo tipo de álgebra lineal entre otras.

Como estas liberías se hacen pensando específicamente en el hardware concreto del computador, son capaces de aprovechar ventajas tanto del compilador como del

propio hardware mucho mejor que las librerías generales. Un ejemplo de estas librerías es la de Intel, denominada Intel-MKL (Math Kernel Library)[13].

En el caso de NEC, tenemos la NEC-NLC(Numeric Library Collection). Una de las muchas funciones que se implementan en estas librerías es la FFT. Concretamente, la NEC-NLC ofrece una librería FFT con la interfaz de la FFTW. La implementación de la FFTW por el equipo de NEC se llama ASLFFTW, y vectoriza mucho mejor que la FFTW dado que está pensada únicamente para el hardware de NEC. Se utilizará como referencia de las mejoras que se propongan en este trabajo. Desgraciadamente, al ser software privado no podremos conocer los detalles de su implementación, pero si que podremos obtener información gracias a los hardware counters.

2.6. Hardware Counters

Los sistemas de cálclo complejo, como por ejemplo los que se usan en los supercomputadores, disponen de lo que se denomina *Hardware Counters*. Como su nombre indica, éstos cuentan multitud de eventos hardware: accesos a memoria, fallos de cache, instrucciones ejecutadas entre otros. Para poder leer su contenido, el usuario puede utilizar instrucciones ensamblador específicas para acceder a la información almacenada de los contadores. Para más comodidad, también se pueden utilizar librerías y herramientas de profiling. Un ejemplo de librería es la librería PAPI(Performance Application Programming Interface)[14]. Ésta proporciona una interfaz unificada entre diferentes sistemas para que el usuario pueda leer los contadores.

En el SX-Aurora, computador sobre el que realiza este proyecto, hay tres maneras de acceder a los contadores hardware[15]. Las dos primeras son utilizando las herramientas PROGINF y FTRACE, y la última es con ensamblador. Las dos primeras generan un sumario con la información de los contadores al acabar el programa. PROGINF recopila los contadores de toda la ejecución, y por lo tanto no es de gran utilidad si se quiere sacar información únicamente de una función, como en nuestro caso. En el trabajo se utilizará la herramienta FTRACE, que separa los contadores por funciones, y ensamblador.

3. Motivación

Debido a la alta presencia de las FFT en multitud de códigos científicos y, por lo tanto, usados en HPC, su rendimiento suele estar en el punto de mira de muchas investigaciones. A esta relevancia se le suma el hecho de que están apareciendo nuevos computadores orientados a HPC con características muy variadas, como por ejemplo *NEC SX Aurora*, introducido de la sección 2.2, que soporta vectores de hasta 256 reales, muy por encima de los procesadores comunes x86 que suportan vectores de hasta 8 elementos.

Si se quiere sacar el máximo rendimiento de los nuevos avances en el hardware, es importante hacer estudios detallados sobre el comportamiento de las aplicaciones y, sobre todo, saber aprovechar los beneficios de las características arquitecturales más novedosas minimizando las posibles caidas de rendimiento.

Este trabajo requiere un metodo muy riguroso. En primer lugar, hay que saber evaluar el rendimiento del kernel específico en máquinas nuevas. Esta evaluación depende de muchos parámetros que implican un conocimiento profundo de la arquitectura, como por ejemplo su jerarquía de memoria.

En segundo lugar hay que dominar los algoritmos de los kernels para poder proponer cambios que resulten en una mejora de rendimiento para las nuevas arquitecturas. Lo ideal es que dichos cambios no sean específicos para una arquitectura, sino que se puedan aplicar a un conjunto de arquitecturas que comparten determinadas características (e.g. vectores largos).

El punto de partida de este proyecto es la observación de que muchos diesñadores de chips orientados a la supercomputación están introduciendo extensiones vectoriales que se benefician del uso de los vectores largos. Además del anteriormente mencionado en la sección 2.2 NEC SX-Aurora, en la extensión vectorial de los procesadores Arm (ARM Scalable Vector Extension[6]) y las instrucciones vectoriales de RISC-V[16] también se aprovechan de los vectores largos.

Estas arquitecturas permiten que códigos que ya parecían estar optimizados al máximo (multiplicaciones de matrices, multiplicaciones vectoriales, FFT, etc...) se pueden reescribir consiguiendo un beneficio de rendimiento. El BSC está trabajando en diferentes proyectos que involucran el desarrollo de microprocesadores basados en arquitectura RISC-V aprovechando su extensión vectorial. El trabajo de evaluación y implementación optimizada de algoritmos FFT para una arquitectura vectorial novedosa y existenta como la de NEC SX-Aurora contribuye a esta clase de proyectos.

El objetivo principal de este proyecto es realizar un estudio exhaustivo y riguroso en diversas arquitecturas de HPC de un kernel muy relevante como el de las FFT. La finalidad de dicho estudio es obtener el conocimiento necesario para poder ajustar el código a la arquitectura, consiguiendo el mejor rendimiento posible.

4. Metodología

En esta sección se explica la metodología utilizada para todos los estudios de este documento.

4.1. Mediciones

Los tiempos de ejecución se han medido en microsegundos, y para obtener suficiente precisión en las mediciones de duración para tamaños de problema pequeños se ha decidido medir siempre 1000 iteraciones.

Además del tiempo de ejecución, se utilizan los MFLOPS como métrica de rendimiento. La razón es evitar las escalas logarítmicas que se requieren para los gráficos que utilizan el tiempo en el eje vertical. Para calcularlos, se ha utilizado la fórmula $\frac{1000*Log_2(N)*N}{time}$. Se basa en que una FFT tiene un coste algorítmico de $N * Log_2(N)$ operaciones y se realizan las 1000 iteraciones anteriormente comentadas.

4.2. Contadores Hardware

Los significados de los contadores hardware que se muestran en las tablas y figuras de este documento son estos:

1. **Instrucciones:** Número total de instrucciones, tanto escalares como vectoriales.
2. **M.Instr:** Millones de instrucciones, tanto escalares como vectoriales.
3. **Ratio de vectorización:** Cociente de las instrucciones vectoriales entre las instrucciones totales.
4. **Longitud Vectorial:** Promedio de las longitud vectorial utilizada por las instrucciones vectoriales.
5. **Ciclos de instrucciones vectoriales aritméticas:** Número de ciclos en que el acelerador ha estado ejecutando instrucciones vectoriales aritméticas.
6. **Ciclos de instrucciones vectoriales de loads:** Número de ciclos en que el acelerador ha estado ejecutando instrucciones vectoriales de lectura de memoria.
7. **Vector Load Elements:** Elementos vectoriales cargados de memoria.
8. **Ratio V.Load.E:** Cociente entre los elementos vectoriales cargados de memoria y los elementos vectoriales totales (cargados y operados).
9. **V.Load MR:** Cociente entre los aciertos y los accesos en la cache de los loads vectoriales.
10. **Cyc/inst vec:** Cociente entre los ciclos empleados en ejecutar instrucciones vectoriales y el número total de instrucciones vectoriales.

4.3. Código fuente

Si se quiere acceder al código fuente de este trabajo, se le puede pedir acceso al repositorio alojado en `https://repo.hca.bsc.es/gitlab/fixers/fft-evaluation` al investigador principal del proyecto.

5. Estudio inicial

En esta sección se expondrá el primer intento de mejorar el rendimiento de la FFT en una máquina con vectores largos. El primer código de referencia es una modificación del algoritmo de FFT proporcionado por el grupo de Programming Models del BSC. Siendo N un número quadrado (i.e con raíz entera) este código se basa en interpretar una FFT de un vector de tamaño N como una FFT de una matriz de dimensiones $\sqrt{N}$ por $\sqrt{N}$. Una vez interpretado el vector como una matriz, se realizan los siguientes pasos:

1. Transponer la matriz
2. Hacer $\sqrt{N}$ FFTs de tamaño $\sqrt{N}$ por filas.
3. Tranponer la matriz
4. Multiplicar todos los elementos por *Twiddle Factors*[1]
5. Realizar las FFTs por filas otra vez
6. Tranponer la matriz

A este esquema de cálculo lo llamaremos "FFT por transposiciones".

La ventaja de este código viene en que realizar $\sqrt{N}$ FFTs de tamaño $\sqrt{N}$ teóricamente se puede acelerar si disponemos de vectores largos, ya que se podrían realizar todas las FFTs en paralelo dentro del mismo vector. De esta manera se quiere aumentar la utilización del vector, es decir, aumentar el número de elementos medio que tienen las intrucciones vectoriales utilizadas. El inconveniente de este esquema para calcular la FFT son las tres transposiciones y la multiplicación de todos los elementos en una de ellas. Aún así, la transposición también es una operación que se puede acelerar utilizando vectores largos.

De entrada no está claro si la implementación de este algorítmo podría encontrar ventajas al ejecutarse en arquitecturas vectoriales. Si se consiguen aprovechar longitudes vectoriales mayores sin aumentar notablemente el número de instrucciones, el beneficio puede ser notable. No obstante, las transposiciones añadidas pueden suponer un sobrecoste muy perjudicial que se deberá evaluar.

[1] Los *Twiddle Factors* son expresiones propias de la FFT, con la forma de tw(k)=$e^{\frac{-j*2*\pi*k}{N}}$

5.1. Primera evaluación

El primer paso ha sido compilar el código en su estado inicial y observar si se cumple la espectativa teórica de rendimiento. El código trae dos esquemas de transposición, uno trivial y otro que transpone la matriz por bloques. A la transposición trivial la llamaremos "Directa"y a la de bloques "Bloques".

En la Figura 6 se compara el rendimiento de una única llamada FFT a la ASLFFTW por referencia, indicada por "1xN", y los dos modelos de FFT por transposiciones. El gráfico superior de la Figura 6 muestra únicamente el tiempo de ejecución, mientras que el inferior muestra una aproximación de los MFLOPS (Millones de operaciones de coma flotante por segundo).

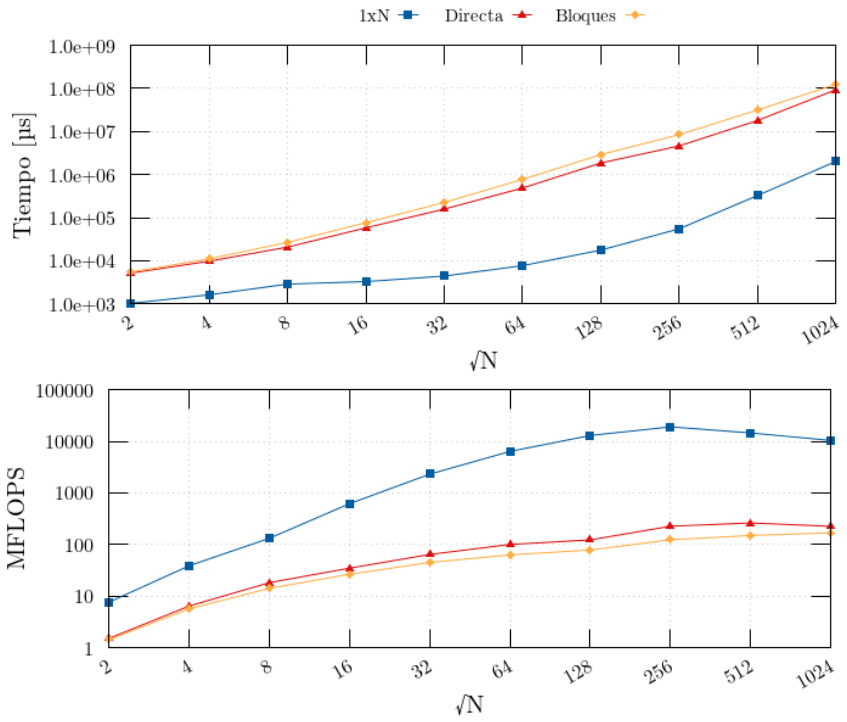

Figura 6: Tiempo de ejecución y MFLOPS de la primera evaluación.

Se puede observar que para cualquier tamaño $\sqrt{N}$, una única llamada de tamaño N consigue un mejor rendimiento que el esquema de FFT por transposiciones.

Para obtener más información sobre esta diferencia, podemos mirar los contadores hardware. En la Tabla 1, los valores de instrucciones y de instrucciones vectoriales se han normalizado respecto a los de realizar una única llamada fft de tamaño N.

Podemos observar como en los esquemas de FFT por transposición se estan realizando muchas más instrucciones totales (hasta 240 veces más para el modelo por bloques). También se realizan más operaciones vectoriales, pero el ratio entre estas y las totales es mucho menor que al realizar una única llamada. A su vez, no se cumple el objetivo de aumentar la longitud media del vector.

	Instrucciones			Ratio de vectorización			Longitud Vectorial		
$\sqrt{N}$	1xN	Directa	Bloques	1xN	Directa	Bloques	1xN	Directa	Bloques
2	1	0.48	0.51	0.00	0.01	0.01	1	1.25	1
4	1	0.83	1.15	0.02	0.03	0.02	1	1.33	1
8	1	1.50	2.00	0.10	0.06	0.04	1	1.31	1
16	1	4.90	6.36	0.04	0.11	0.08	16	1.18	1
32	1	13.18	18.33	0.08	0.17	0.12	32	1.16	1
64	1	35.62	53.07	0.15	0.23	0.15	64	1.14	1
128	1	94.62	146.60	0.25	0.29	0.18	128	1.12	1
256	1	117.04	253.27	0.35	0.07	0.03	256	14.86	16
512	1	100.62	242.86	0.45	0.07	0.02	256	19.80	22.4
1024	1	92.17	240.94	0.50	0.06	0.02	256	26.47	32

Tabla 1: Contadores hardware de los tres esquemas para diferentes tamaños de $\sqrt{N}$.

Recordemos que el tamaño máximo de esta arquitectura es de 256 elementos, y estamos llegando a un máximo de 32 con los esquemas propuestos.

Para entrar aún más en detalle, miraremos los contadores hardware de las diferentes etapas de ambos esquemas por transposiciones. Se ha separado el esquema en tres etapas: las dos llamadas a $\sqrt{N}$ FFTs de tamaño $\sqrt{N}$, que indicaremos con la notación "$2 \times$ FFT"; las dos transposiciones normales indicadas por "$2 \times$ TP" y la transposición que incluye la multiplicación por los *Twiddle Factors*, indicada por "$1 \times$ TP_W".

Con esta información podremos entender mejor la razón por la que los dos esquemas de FFT por transposición tienen un peor rendimiento que la única llamada de tamaño N y, más concretamente, por que la versión que transpone de manera directa obtiene mejores resultados que la de bloques. Hay dos factores principales que nos estan perjudicando: el sobrecoste de las transposiciones y la poca vectorización del código.

Sobrecoste de las transposiciones

En los gráficos de la Figura 7 observamos el ratio de instrucciones que supone cada etapa respeto al total. A la izquierda tenemos el gráfico que corresponde a la versión por bloques y a la derecha a los de la transposición directa. Vemos que a medida que el tamaño del problema crece, la mayor parte de las instrucciones se situan en las transposiciones (en el caso de bloques, las FFTs acaban significando solo un 8 % del total de instrucciones. Este análsis nos indica que el sobrecoste de las transposiciones está eclipsando el posible beneficio que teóricamente otorgaría realizar muchas FFTs de tamaño más pequeño en un vector.

Podemos observar como el hecho de que la transposición directa rinda mejor se relfeja en que la proporción de instrucciones correspondiente a las transposiciones es menor, comparado con la transposición por bloques.

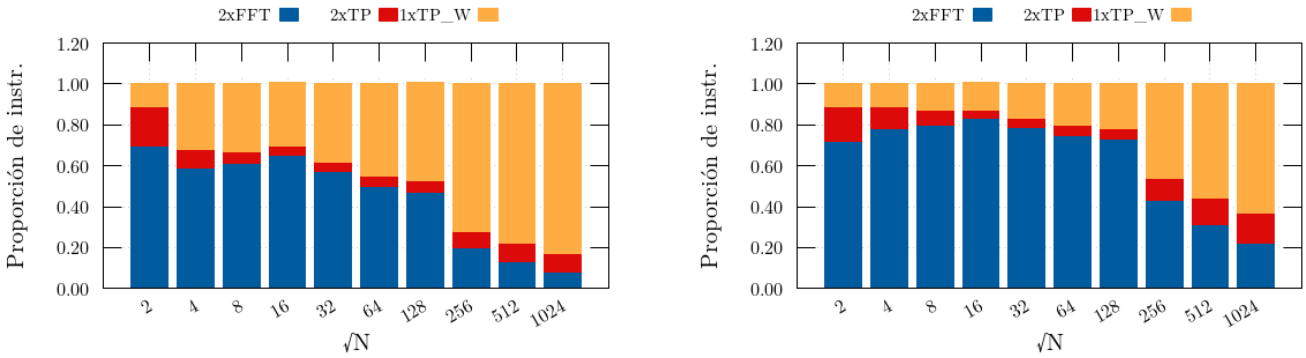

Figura 7: Millones de elementos transpuestos por segundo siguiendo la transposición de bloques (izquierda) y directa (derecha).

Vectorización

En la Tabla 2 observamos la vectorización de las etapas presentadas anteriormente. En primer lugar, las múltiples FFTs no estan cumpliendo nuestra espectativa de vectorización. Podemos ver en la última columna como la longitud de vector promedio solo llega hasta 32, para el tamaño $\sqrt{N} = 1024$. Recordemos que la idea era realizar muchas ffts de tamaño pequeño en vez de una fft grande con la esperanza de realizarlas en paralelo dentro de un mismo vector, aumentando la vectorización.

	Vectorización / (Longitud Vectorial)				
	2xTP		1xTP_W		2xFFTs
$\sqrt{N}$	Bloques	Directa	Bloques	Directa	Ambas
2	0/-	0.01/(2)	0/-	0/-	0.01/(1)
4	0/-	0.02/(4)	0/-	0/-	0.03/(1)
8	0/-	0.04/(8)	0/-	0/-	0.07/(1)
16	0/-	0.07/(8)	0/-	0/-	0.13/(1)
32	0/-	0.09/(8)	0/-	0/-	0.21/(1)
64	0/-	0.10/(8)	0/-	0/-	0.31/(1)
128	0/-	0.10/(8)	0/-	0/-	0.39/(1)
256	0/-	0.10/(8)	0/-	0/-	0.15/(16)
512	0/-	0.10/(8)	0/-	0/-	0.19/(24)
1024	0/-	0.10/(8)	0/-	0/-	0.23/(32)

Tabla 2: Ratio de vectorización y longitud vectorial de los esquemas de transposiciones por bloques y directa, separado entre las dos transposiciones normales, la que multiplica por los *Twiddle Factors* y las 2 FFTs.

En segundo lugar, las transposiciones no se estan auto-vectorizando eficientemente. En el caso de bloques no se vectorizan en absoluto; podemos ver en la Tabla 2 como la lontigud vectorial promedia es 0. En el caso directo solo se vectoriza muy ligeramente la transposición sin multiplicación (tercera columna), pero sin superar los 8 elementos, quedándose muy lejos de la longitud máxima de 256 elementos.

Una vez identificados los problemas de la versión inicial del código, pasamos a mejorarlos por separado.

5.2. Múltiples FFTW

El primer problema que abordaremos es optimizar las $\sqrt{N}$ FFTs de tamaño $\sqrt{N}$. Por suerte, la librería fftw (y su versión en NEC, la ASLFFTW) dispone de una función llamada `fftw_plan_many_dft`. Esta función hace justo lo que necesitamos; prepara una configuración que hace múltiples FFT dentro de una única array. En el contexto de la FFTW, esta configuración recibe el nombre de "plan", y concretamente es el conjunto de los datos precomputados para acelerar el código de la FFT y los parámetros para ejecutarla, tal y como se explica en la sección 2.4. De ahora en adelante, cualquier código que se base en utilizar esta función obtendrá el nombre de "many" en las leyendas de las tablas y gráficos.

Para poder evaluar esta función por separado, se ha implementado un pequeño programa para comparar el tiempo que requiere realizar una FFT de tamaño N, indicado con "$1 \times$ N; $\sqrt{N}$ llamadas FFT de tamaño $\sqrt{N}$, indicado con "$\sqrt{N} \times \sqrt{N}$" y la llamada *many*.

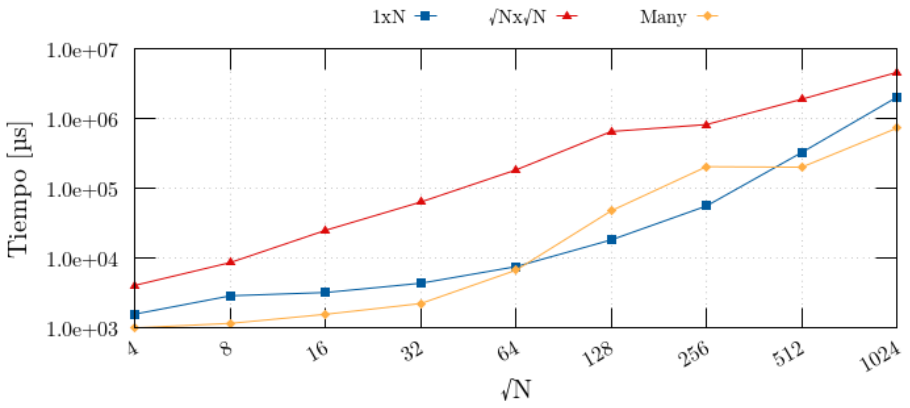

Figura 8: Tiempos de las diferentes formas de llamar a la librería ASLFFTW.

Para entender la Figura 8 hace falta remarcar que para un punto $\sqrt{N}$ del eje horizontal, la línea azul representa la FFT con tamaño N mientras que la roja y la amarilla realizan $\sqrt{N}$ FFTs de tamaño $\sqrt{N}$. El número de elementos operados es el mismo, pero repartidos en una o varias FFTs.

Podemos observar como hacer múltiples FFTs en llamadas distintas (la línea roja) rinde peor que hacer una única llamada más grande, confirmando los resultados de la sección anterior. En cambio, utilizar la función *many* requiere menos tiempo que utilizar la llamada más grande. Cabe destacar que utilizar solo la llamada *many* no devuelve el resultado correcto; para que éste lo fuese se requiere del esquema de las transposiciones.

Al igual que antes, también se han obtenido los contadores hardware relevantes:

	Instrucciones			Ratio de vectorización			Longitud Vectorial		
$\sqrt{N}$	1xN	$\sqrt{N}$x$\sqrt{N}$	Many	1xN	$\sqrt{N}$x$\sqrt{N}$	Many	1xN	$\sqrt{N}$x$\sqrt{N}$	Many
4	1.00	1.84	0.46	0.13	0.03	0.03	1.00	1.00	4.00
8	1.00	1.78	0.23	0.31	0.07	0.07	1.00	1.00	8.00
16	1.00	7.21	0.45	0.15	0.13	0.13	16.00	1.00	16.00
32	1.00	14.24	0.45	0.23	0.21	0.21	32.00	1.00	32.00
64	1.00	28.04	0.44	0.32	0.31	0.31	64.00	1.00	64.00
128	1.00	55.66	0.44	0.40	0.39	0.39	128.00	1.00	128.00
256	1.00	35.84	0.44	0.51	0.15	0.49	256.00	16.00	256.00
512	1.00	17.34	1.45	0.50	0.19	0.58	256.00	22.40	69.32
1024	1.00	10.16	0.80	0.51	0.23	0.58	256.00	32.00	131.00

Tabla 3: instrucciones, vectorización y longitud vectorial de diferentes llamadas de la librería ASLFFTW

En la tercera columna de la Tabla 3 (Instrucciones $\sqrt{N} \times \sqrt{N}$) se ve que realizar múltiples FFTs en llamadas distintas conlleva muchas más instrucciones, hasta 55 veces más para $\sqrt{N}$=128. A su vez, también resulta en utilizar vectores más cortos y un ratio de vectorización menor.

En cambio, utilizar la función *many* reduce el número de instrucciones y aumenta ligeramente el porcentage de vectorización. Los casos $\sqrt{N}$=512 y $\sqrt{N}$=1024 son curiosos, ya que baja el tamaño del vector medio y aumenta el número de instrucciones respeto a los otros tamaños.

No obstante, como veíamos en la Figura 8, los casos en los que la función *many* está tardando más que una única llamada no son 512 y 1024, sino 128 y 256. Mirando tan solo estos contadores podemos asegurar que la explicación de este hecho no está relacionada con el número de instrucciones o la vectorización en sí, ya que para estos tamaños los contadores son positivos: la llamada *many* hace menos instrucciones y vectoriza más.

Aunque descubrir la razón por la que estos dos tamaños se comportan peor que los demás tampoco afecta directamente a nuestro código, ya que no tenemos acceso para modificar la función *many*, se ha decidio explorar los otros contadores en busca de respuestas.

En la izquierda de la figura 5.2 se encuentra a modo de histograma los ciclos que corresponden a instrucciones vectoriales aritméticas y de memoria. En la derecha se muestra el contador de elementos vectoriales cargados de memoria. En ambos casos se han normalizado los valores a los de la única llamada de tamaño N, ya que es la referencia que queremos superar.

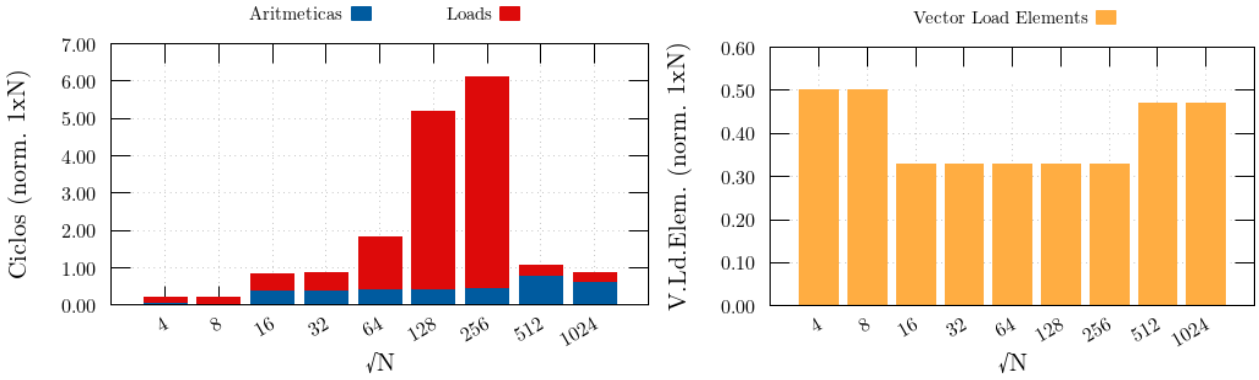

Figura 9: Ciclos de instrucciones vectoriales aritméticas y de memoria (izquierda) y elementos vectoriales cargados de memoria (derecha), ambos de la llamada *many* normalizados a la llamada de una única FFT.

Podemos observar como en los tamaños problemáticos, 128 y 256, la versión *many* emplea más ciclos en instrucciones vectoriales que la versión de una única llamada, hasta 6 veces más. Si miramos los ciclos de las instrucciones vectoriales aritméticas (en azul), vemos que siguen en la línea de los demás tamaños, empleando menos ciclos en ellas que la referencia (valores por debajo de 1.0).

En cambio, si miramos los ciclos asociados a las instrucciones vectoriales que acceden a memoria, en rojo, podemos ver como la versión *many* emplea muchos más ciclos en ellas que la versión de única llamada. Ésto podría ser consecuencia de un mayor número de elementos leídos y escritos en memoria o bién de loads más lentos.

Con el gráfico de la derecha de la Figura 5.2 vemos que los elementos tratados en instrucciones vectoriales siguen en la misma línea que para los demás tamaños y son siempre menos que la referencia, ya que el valor normalizado es menor a 1.0. Por lo tanto, podemos atribuir el peor rendimiento a que las instrucciones vectoriales de memoria están tardando más.

Utilizando la herramienta FTRACE, presentada anteriormente en la sección 2.5, se ha obtenido más información de los contadores hardware. A continuación se muestra un extracto del sumario que genera la herramienta para el tamaño 256:

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1000	0.203(77.3)	0.203	11089.5	8469.7	99.63	256.0	0.203	0.002	0.169	100.00	wrapper_many
1000	0.058(21.9)	0.058	92100.4	66713.5	99.65	256.0	0.053	0.003	0.012	100.00	wrapper_1xn

Figura 10: Output de la herramienta FTRACE con $\sqrt{N}$=256

Nos fijamos en la antepenúltima columna de la figure 10, la columna ”CPU_PORT_CONF”. En ésta se muestran los segundos que ha estado la función (indicada en la última columna) en ”Conflictos de puerto de CPU”. Podemos ver que si restamos este tiempo al ”AVER.TIME” de la función, obtendríamos el rendimiento esperado: la función *many* tardando menos que la 1xN.

Para assegurar más la relación entre este contador y el hecho de que la función *many* tarde más en los tamaños 128 y 256, se ha utilizado la herramienta FTRACE con un tamaño de 512, un tamaño en el cual la llamada *many* tarda menos que la llamada única. Como podemos ver en la Figura 11, el contador de segundos de conflictos de puerto de CPU es inferior en la llamada *many*. Aquí confirmamos la correlación de que si los conflictos de puerto de CPU son pocos, la llamada *many* tarda menos que la llamada única; y en el caso contrario tarda más.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD HIT	LLC E.%	PROC.NAME
1000	0.325(61.4)	0.325	79659.1	55144.4	99.64	256.0	0.304	0.016	0.095	99.01		wrapper_1xn
1000	0.200(37.8)	0.200	59135.5	40879.3	99.07	69.3	0.194	0.003	0.051	100.00		wrapper_many

Figura 11: Output de la herramienta FTRACE con $\sqrt{N}$=512

No se ha podido encontrar más información sobre la naturaleza de los conflictos de puerto de la CPU ni información sobre como solucionarlos. Dado que tampoco se puede modificar el código de la librería ASLFFTW, no se ha avanzado más en la investigación de estos dos casos. Se ha concluido que la razón de su bajo rendimiento son unas instrucciones de acceso a memoria anómalas que requerien más ciclos de lo habitual y que la anomalía parece estar relacionada con conflictos de puerto de la CPU.

5.3. Transposiciones vectoriales

El siguiente paso para mejorar el esquema de FFT por transposiciones es mejorar el rendimiento de las transposiciones. Si recordamos la Figura 7, las transposiciones componían gran parte del número de instrucciones totales y no estaban vectorizadas correctamente.

Se ha revisado la librería matemática de NEC, la NLC, buscando funciones que implementen una transposición de una matriz compleja y no se han encontrado. Transponer una matriz compleja no es igual de trivial que una matriz de enteros o reales. Los elementos de las instrucciones vectoriales son de 64 bits, y por lo tanto se corresponden a un double. Un complejo esta compuesto de dos doubles (la parte real y la parte imaginaria), y por lo tanto requiere de 128 bits.

5.3.1. Transposición con *Strided Loads*

Una manera de transponer una matriz de enteros o doubles utilizando instrucciones vectoriales es utilizando *Strided Loads* y *Stores*. Un load con stride es aquel que a partir de una memoria M, una dirección base D y un stride K llena los elementos(i) de un vector tal que $M = D + K * i$. Se realiza un load con un stride igual a la dimensión horizontal de la matriz para cargar la primera columna y luego un store convencional para guardarla en la primera fila de la matriz transpuesta. Este procedimiento se repite para todas las columnas, transponiendo la matriz.

Se puede observar un esquema de este método en la Figura 12

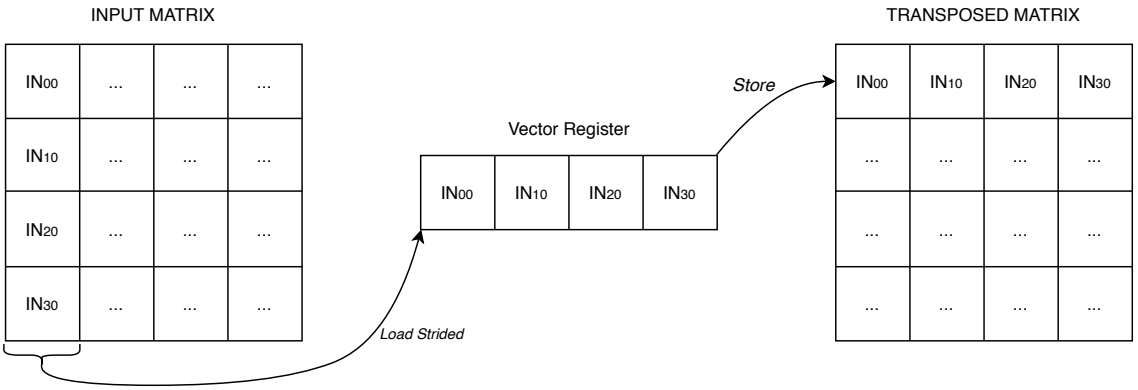

Figura 12: Esquema de transposición con *Strided Loads* de una matriz no compleja por columnas

La transposición de una matriz compleja no permite utilizar este método sin realizarle antes unas modificaciones. Debemos interpretar la matriz de complejos de tamaño N^2 como una matriz de doubles de tamaño $N \times (2 \times N)$. En las columnas pares encontramos la parte real de los números y en las impares la parte imaginaria.

Siendo *(F,C)* la fila y columna de la matriz, para transponerla de manera similar a la anterior deberíamos llenar un vector con los elementos [(0,0), (0,1), (1,0), (1,1), (2,0), (2,1), ...] tal como se ve en la Figura 13. Este vector luego lo deberíamos guardar en la primera fila de la matriz transpuesta al igual que antes y repetir este proceso para todas las columnas pares de la matriz original.

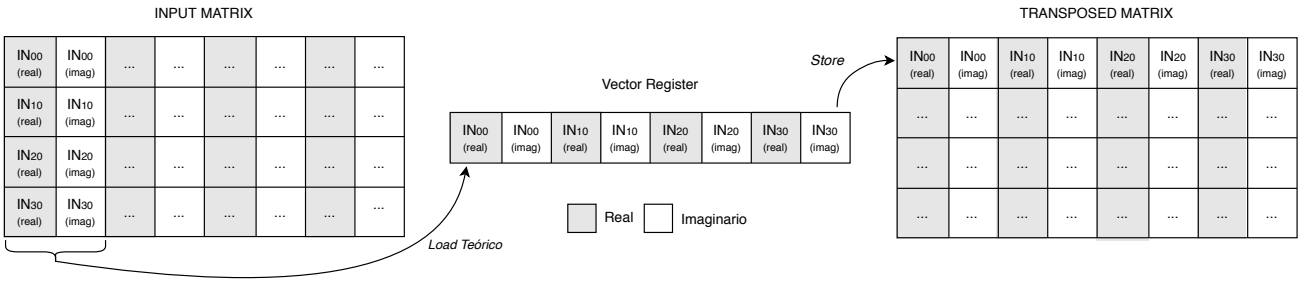

Figura 13: Esquema de transposición de una matriz compleja por columnas

En la Figura 13 se ha marcado la lectura como "Load Teórico", ya que en la arquitectura del SX-Aurora no existe una manera directa de llenar un vector con los valores entrelazados de la parte real e imaginaria de cada columna de la matriz compleja (es decir, leer dos elementos doubles antes de aplicar el stride para saltar de fila).

La primera implementación que se ha relizado en este proyecto para transponer matrizes complejas de manera vectorial se basa en cargar la parte real e imaginaria

de los elementos en vectores distintos. En primer lugar se carga la fila par (real) en un vector y seguidamente la segunda fila, la impar (imaginaria) en otro; utilizando loads con stride al igual que en el método de la Figura 12.

La diferencia en comparación con el método convencional es que los stores que se utilizan son con stride, concretamente con stride igual al tamaño de dos doubles. De esta manera, se guarda el vector que contiene la columna de los elementos reales en las posiciones pares de la primera fila transpuesta y el vector de los imaginarios en las posiciones impares; se entrelazan los dos vectores en memoria. Esta implementación se puede ver esquematizada en la Figura 14.

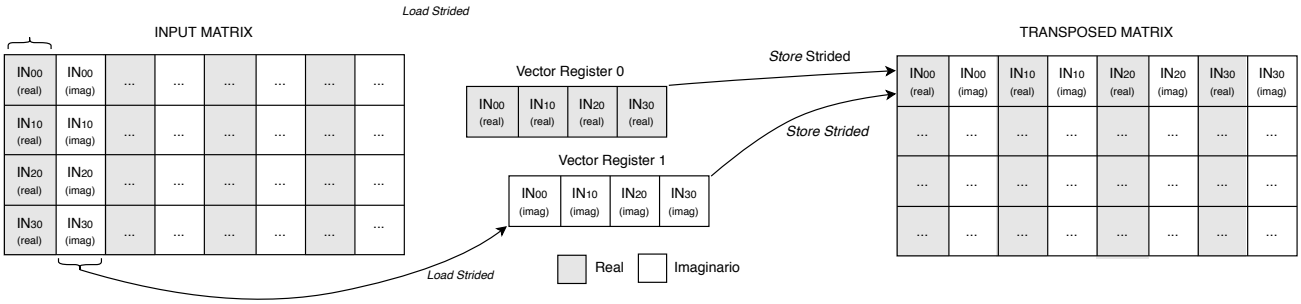

Figura 14: Esquema de la implementación con *Strided Loads* por columnas

Esta implementación tiene el inconveniente de que incluso teniendo $N \times (2 \times N)$ doubles en la matriz, la longitud máxima que utilizaremos es la de una columna, N. Asumiendo que toda la columna cabe en un vector, requeriremos de $2 * N$ loads y $2 * N$ stores, por lo tanto $4 * N$ instrucciones.

Por último, recordamos que el esquema de FFT por transposiciones requiere que en una de ellas se multipliquen todos los elementos por determinados valores. Debido a la naturaleza de la multiplicación de números complejos, tener los elementos reales e imaginarios por separado facilita mucho la multiplicación.

Siendo $z0 = a + bi$ y $z1 = c + di$, $z0 * z1 = (a * d - b * c) + (a * d + b * c) * i$, tanto la parte real como la parte imaginaria requerirían de 2 multiplicaciones y una suma o resta. Utilizando una instrucción del tipo FMADD[2], podemos reducir el número de instrucciones a dos para la parte real y dos para la imaginaria.

Se ha implementado una transposición con multiplicación que requiere de los mismos load y stores que antes pero añadiendo 2 loads más (para la parte real e imaginaria de los *Twiddle Factors*) y 4 operaciones aritmeticas para la multiplicacion. Por lo tanto, se requieren $(4 + 2 + 4) * N = 10 * N$ instrucciones.

5.3.2. Transposición con *Gathers*

Esta implementación tiene como objetivo maximizar la longitud del vector. La intención de este método es cargar toda la matriz en un vector, ordenado de una

[2]Fused multiply add: realiza en una única instrucción una multiplicación y una suma

manera concreta, y luego realizar un único store para guardarlo en la matriz transpuesta.

La arquitectura cuenta con una instrucción que nos permite cargar elementos de posiciones totalmente arbitrarias de memoria, la instrucción *Gather*. Requerimos de un vector en el que cada elemento corresponde a una dirección de memoria. Siendo I el indice del vector, Vd el vector resultado y Va el vector con las direcciones de memoria, esta instrucción realiza la siguiente operación: $Vd[I] = M[Va[I]]$. Al vector con las direcciones de memoria le llamaremos vector de índices.

Ésto nos permite preparar unos índices de antemano que servirán para cualquier matriz de un mismo tamaño. Estos índices corresponden al patrón expuesto en la Figura 13 pero sin limitarse a las dos primeras columnas.

Para transponer una matriz, cargamos estos indices con un load convencional en un vector, le sumamos la dirección base de la matriz a transponer, realizamos el gather y luego guardamos el vector resultante en la dirección de la matriz transpuesta. Con 4 instrucciones (1 de ellas siendo una simple suma y por lo tanto con menor latencia que las que acceden a memoria) hemos transpuesto toda la matriz, suponiendo que toda quepa en un registro vectorial. De no ser así, se transpone la matriz a bloques.

No se ha implementado la versión de la transposición que también multiplica los elementos por los *Twiddle Factors*. Al tener la parte real e imaginaria en el mismo vector, la multiplicación ya no es trivial y complicaría demasiado la implementación, añadiendo muchas instrucciones.

5.3.3. Evaluación de ambos métodos

En la evaluación, la versión que transpone utilizando *Strided Loads* se indicará con el nombre "S.Loads" y la que utiliza *Gathers* como "Gathers".

En primer lugar observamos el tiempo de ejecución de la transposición sin la multiplicación por los *Twiddle Factors*. En la Figura 15 observamos la comparación tanto con tiempos como con millones de elementos transpuestos por segundo entre la transposición escalar y las vectoriales S.Loads y Gathers.

Comprobamos que las versiones vectoriales rinden notablemente mejor que la escalar. En la parte superior de la Figura 15 podemos ver como el tiempo de las versiones vectoriales es menor que la escalar, y en la parte inferior podemos ver como éstas transponen muchos más millones de elementos por segundo. Para un tamaño de 16x16, la versión con *Gathers* llega a los casi 4000 elementos por segundo mientras la referencia escalar esta sobre los 350. También es notable que la versión con *Gather* rinde mucho mejor que la de los *Strided Loads* hasta el tamaño 16x16, a partir del cual se acaban igualando en 128x128. Para explicar este comportamiento, estudiaremos los contadores hardware de ambas versiones.

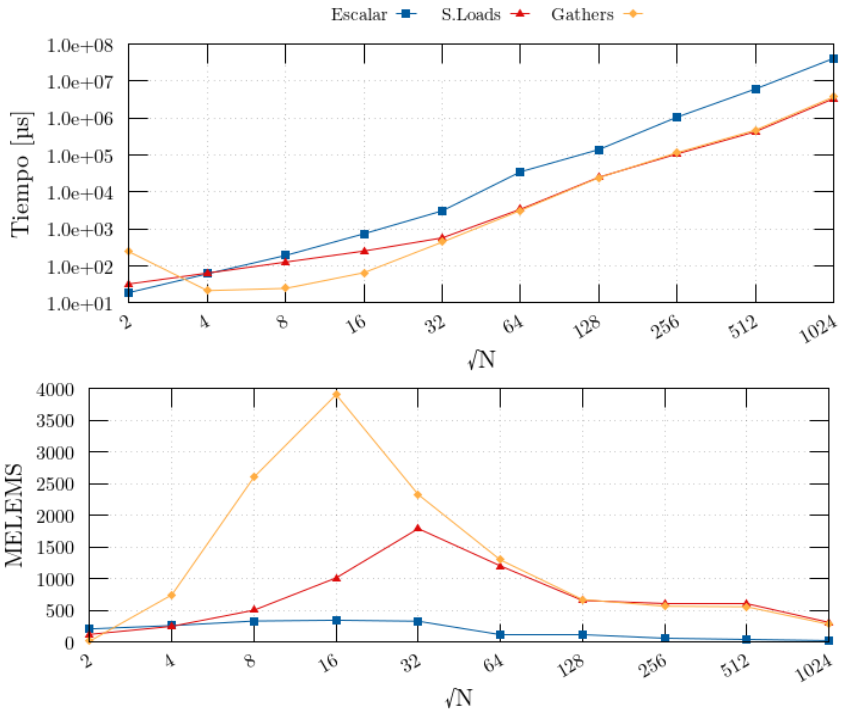

Figura 15: Tiempos y millones de elementos por segundo de las transposiciones.

	M. Instrucciones		Ratio de vectorización		Longitud Vectorial	
$\sqrt{N}$	S.Loads	Gathers	S.Loads	Gathers	S.Loads	Gathers
2	0.06	0.02	0.13	0.23	2	8
4	0.11	0.02	0.15	0.23	4	32
8	0.20	0.02	0.16	0.23	8	128
16	0.40	0.04	0.16	0.27	16	256
32	0.78	0.13	0.16	0.30	32	256
64	1.55	0.52	0.17	0.31	64	256
128	3.08	2.05	0.17	0.31	128	256
256	6.16	8.20	0.17	0.31	256	256
512	19.98	32.77	0.21	0.31	256	256
1024	70.67	131.08	0.23	0.31	256	256

Tabla 4: Contadores hardware de las transposiciones vectoriales.

La Tabla 4 nos muestra que la transposición con *Gathers* consigue una longitud vectorial igual a $N \times (2 \times N)$; igual a los elementos de 64 bits que tiene la matriz. A partir de la matriz de 16x16 complejos la transposición ya llega a la longitud vectorial máxima y cada vez que se dobla la dimensión de la matriz se cuadruplica el número de instrucciones. Hasta el punto de 16x16 el algoritmo es constante en instrucciones, tal y como se ha explicado en su diseño.

En la transposición con *Strided Loads*, se cumple el límite de la longitud vectorial impuesto por el numero de filas de la matriz. Observamos que eso le permite no saturar la longitud vectorial hasta el tamaño 256x256, donde tiene un número de millones de instrucciones inferior que la versión con *Gathers*, que lleva cuadruplicando sus instrucciones desde tamaños inferiores. A partir del tamaño 256x256, ambas siguen cuadruplicando instrucciones aunque la versión con *Strided Loads* empieza con un valor menor.

La transposición con multiplicación tiene un rendimiento ligeramente inferior a la transposición normal. En la Figura 16 se muestra la referencia escalar, la transposición con multiplicación por los Twidle Factors y, por tal de exponer el sobrecoste de las multiplicaciones, se ha mostrado de nuevo la transposición vectorial con *Strided Loads* sin multiplicación.

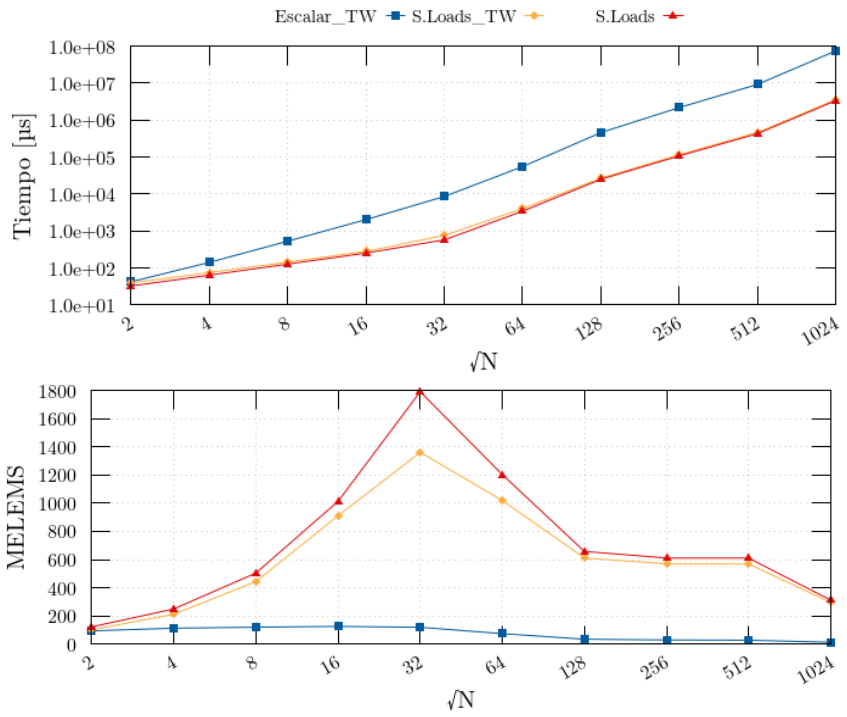

Figura 16: Tiempos y millones de elementos por segundo de las transposiciones con *Twiddle Factors*.

Podemos concluir que la versión vectorial con multiplicación sigue siendo mucho más eficiente aunque haya una disminución de rendimiento en comparación con la transposición vectorial sin multiplicación.

5.4. Evaluación final del esquema de FFT por transposiciones

Finalmente, se juntan las mejoras en el cálculo de múltiples FFTW de la sección 5.1 con las mejoras en las transposiciones de la sección 5.2 con el fin de intentar mejorar el esquema de la FFT por transposiciones.

Antes de juntar los diferentes algoritmos y medir el rendimiento, se ha decidido modelizar el problema. En la Figura 17 se muestra el tiempo de una única llamada de tamaño N, indicada con "1 × N"; el tiempo agregado de hacer dos llamadas de la función *many*, indicado con "2 × Many"; el tiempo de dos transposiciones normales y una con multiplicación agregado, indicado con "Transposiciones", y por último la suma de estos dos últimos valores, obteniendo la predicción de tiempo del esquema de FFT por transposiciones que se indica con "Predicción".

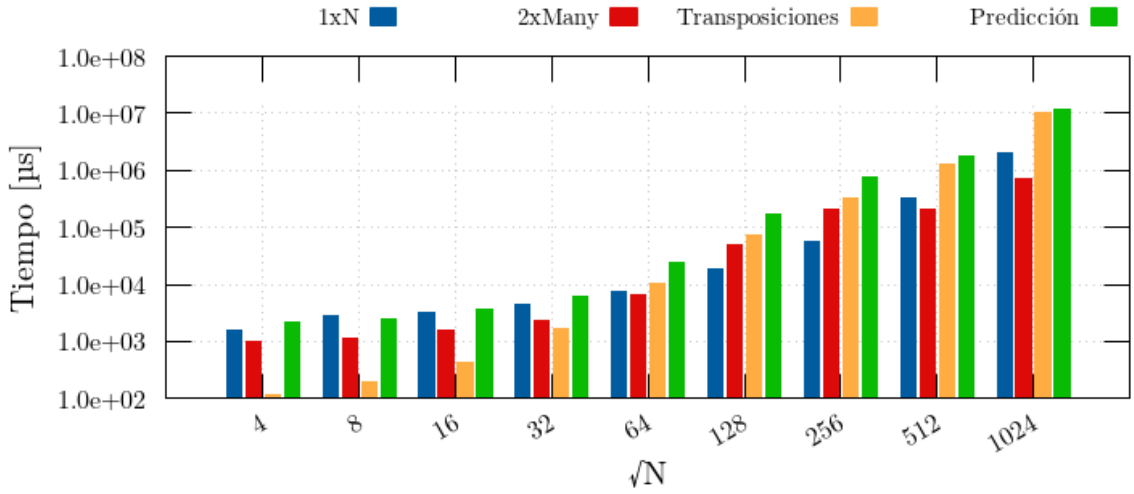

Figura 17: Modelo de tiempo del esquema por transposiciones.

Es importante recalcar que en la Figura 17 el eje vertical es logarítmico y las columnas verdes son la suma de las rojas y naranjas.

En la Figura 17 podemos observar dos conclusiones negativas. La primera es que el tiempo agregado de realizar dos llamadas *many* es muy similar al de realizar una normal. Para los casos estudiados en la sección 5.1, los tamaños 128 y 256, hacer dos llamadas *many* tarda alrededor de un orden de magnitud más que una única llamada normal. Incluso descartando estos tamaños, las transposiciones deberían ser muy eficientes por tal de que al añadirlas al tiempo de dos llamadas *many* se consiga mejor rendimiento en los demás tamaños.

La segunda conclusión no augura que ésto último se vaya a cumplir. Podemos ver como a partir del tamaño 64, las transposiciones ya tardan notablemente más que la única fft.

El único tamaño en el que parece que podríamos encontrar una mejora de rendimiento es para $\sqrt{N}$=8.

Para comprobarlo, en la Figura 18 podemos ver el tiempo medido a la izquierda y el tiempo normalizado a la única llamada normal a la derecha. La FFT realizada con única llamada a la librería que se utiliza de referencia se sigue indicando con "1 × N", el cálculo de la FFT utilizando el esquema por transposiciones propuesto se ha indicado con "FFT Transp" y la predicción con "Predicción". Esta última se ha añadido para validar la precisión de la predicción de la Figura 17

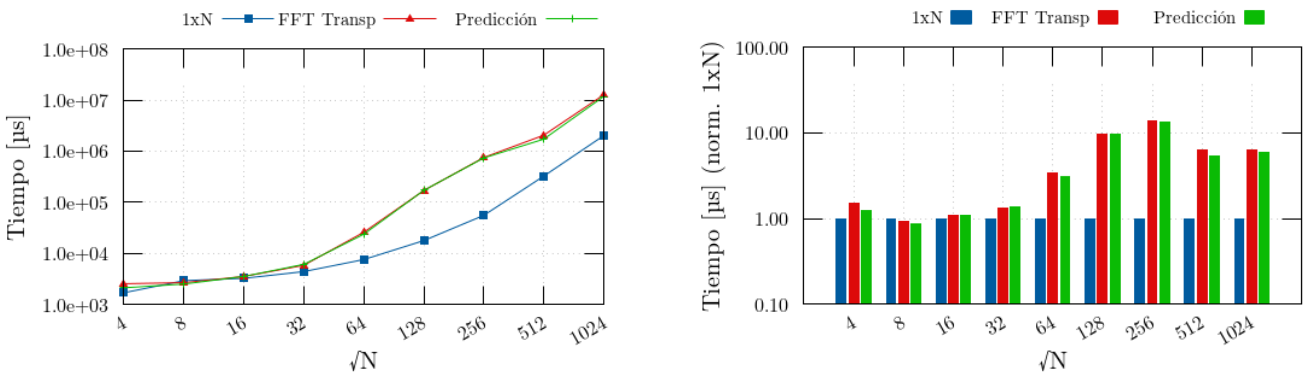

Figura 18: Tiempo medido del esquema por transposiciones (izquierda) y su normalización a ASLFFTW(derecha).

Comprobamos que el modelo se ajusta a la realidad y el único caso en el que tenemos una ligera mejora es para el tamaño $\sqrt{N}$=8, tal y como se había previsto. Concretamente, obtenemos speedup de 1.09x.

Para completar el estudio del esquema de FFT por transposiciones, miraremos los contadores hardware.

	M. Instrucciones		Ratio de vectorización		Longitud Vectorial	
$\sqrt{N}$	1xN	FFT Transp	1xN	FFT Transp	1xN	FFT Transp
4	2.34	2.46	0.13	0.05	1	6.46
8	5.22	2.80	0.31	0.09	1	12.58
16	5.18	5.47	0.15	0.14	16	22.09
32	7.57	8.30	0.23	0.22	32	42.00
64	11.90	13.78	0.32	0.30	64	78.77
128	21.11	26.45	0.40	0.37	128	144.84
256	36.93	53.70	0.51	0.39	256	256.00
512	183.18	597.06	0.50	0.55	256	79.79
1024	762.28	1444.98	0.51	0.54	256	142.84

Tabla 5: Comparación de los contadores hardware de la referencia y el esquema de FFT por transposiciones.

En la Tabla 5 observamos como el único caso en que la versión con transposiciones realiza menos millones de instrucciones es cuando la raíz de N es 8. Para tamaños grandes, se exagera la diferencia de millones de instrucciones. También podemos ver que la versión con transposiciones consigue vectores más largos excepto en los dos últimos tamaños. Estos dos últimos casos eran esperados si se revisa la Tabla 3, referente a la función *many*.

Con estos resultados se da por finalizado el estudio del esquema de FFT por transposiciones. Al diseñar el esquema, se esperaba que la función *many* aprovechase en mayor medida la longitud vectorial. Se esperaba que para realizar P transformadas de tamaño Q se utilizaría una longitud superior a la que se utiliza para una transformada de tamaño $P*Q$, pero hemos comprobado en la Tabla 3 que esto no se cumple.

Si al hecho de que la función *many* no sea tan óptima como esperábamos se le añade el coste de las transposiciones, nos encontramos con que es más eficiente realizar una única llamada a la librería ASLFFTW en vez de utilizar el esquema propuesto.

6. Implementación de una FFT vectorial

Hemos visto que con el esquema de la FFT por transposición solo hemos conseguido una mejora para un tamaño particular, y no muy significativa. Hemos remarcado que uno de los factores que limita la optimización es que la llamada *many* de la librería ASLFFTW no es tan óptima como se esperaba. Llegados a este punto, se ha decidio implementar un algorítmo que calcule la FFT utilizando las instrucciones vectoriales de la arquitectura NEC SX-Aurora.

6.1. Entiendiendo la FFT

El primer paso para implementar una FFT es partir de la DFT, ya que la primera es una optimización de la última. Si $X = DFT(x)$, la DFT se puede describir con la siguiente ecuación:

$$X[k] = \sum_{n=0}^{N} x[k] * W_N^{(k*n)} \ [3]$$

Como se realiza el sumatorio para todos los N elementos de X, tenemos N sumatorios de N elementos y por lo tanto un coste de N^2. La función W, a la que también nos hemos referido como *Twiddle Factors* durante este trabajo, presenta mucha periodicidad. Aprovechando la periodicidad de ésta, se puede optimizar la DFT hasta llegar a la FFT. En la bibliografía se encuentra un recurso audiovisual[17] que explica de manera clara el concepto de la FFT y el algoritmo de Cooley-Tukey[18] que se explicará a continuación.

Resumidamente, hay dos maneras de simplificar la DFT. La primera se llama *decimation in time* y la segunda *decimation in frequency*. Ambas se basan en descomponer la DFT en DFTs más pequeñas.Estas DFTs más pequeñas pueden ser del tamaño que queramos, hasta llegar a una DFT de dos elementos. A continuación, en la Figura 19, se muestra un ejemplo en la que se realiza un prototipo de FFT de 8 elementos realizando dos DFT de tamaño 4.

[3]Recordemos que $W_b^a = e^{\frac{-j*2*\pi*a}{b}}$

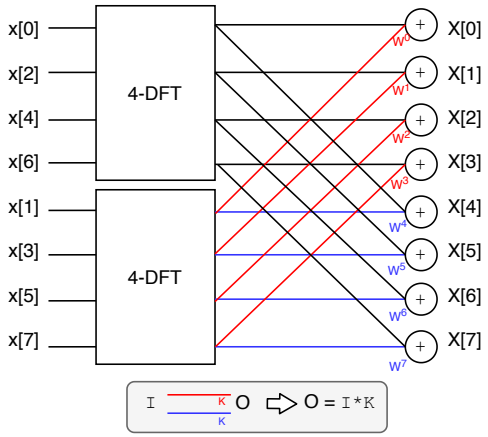

Figura 19: DFT de 8 elementos interpretada como dos DFT de 4 elementos.

Primero se desordena la entrada. Si se tratase de una *decimation in frequency*, la reordenación de los elementos se haría al final. En el caso de realizar una única división (como en nuestro caso, que realizamos una DFT de 8 elementos utilizando 2 de 4 elementos), tan solo hace falta separar las entradas en pares e impares. Una vez hecho esto, realizamos las dos DFTs pequeñas y al acabar entrelazamos los resultados, multiplicandolos por un determinado Twiddle Factor y sumándolos entre si.

Las multiplicaciones siguen un patrón comúnmente denominado *Butterfly* o *Mariposa*. En nuestro ejemplo, solo realizamos una única *Mariposa*. Cada elemento de la mitad inferior de la *Mariposa* se multiplica por dos *Twiddle Factors*[4]. En la Figura 19 se han marcado con color negro, rojo y azul los valores que solamente se suman, se multiplican por el primer Twiddle Factor y por el segundo respectivamente.

Si seguimos realizando reordenaciones y divisiones en DFTs parciales menores, acabamos con el siguiente esquema:

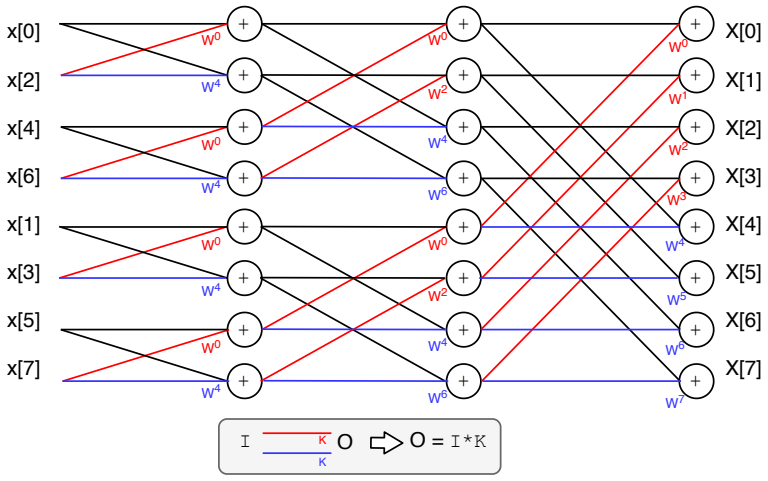

Figura 20: DFT de 8 elementos dividida al máximo.

[4]En concreto, se multiplica por un W_N^k y un $W_N^{k+N/2}$

En la Figura 20 se puede observar como se generan $Log_2(N)$ fases. En cada una de ellas estamos realizando N sumas y N multiplicaciones. Por lo tanto, el coste del algorítmo tiene el orden de $N * Log_2(N)$. En la primera fase se realizan 4 *Mariposas*, en la segunda dos y en la última una. Visto de otra manera, en cada fase se realizan las *Mariposas* de las parejas de *Mariposas* de la fase anterior.

6.2. Primera implementación: FFTP 1

La primera implementación de la FFT realizada en este proyecto recibe el nombre de *FFTP 1*. Esta implementación traduce el esquema de la Figura 20 a código vectorial. El algoritmo itera por las $Log_2(N)$ fases. En cada fase se siguen las mismas acciones:

1. Reordenar los resultados utilizando instrucciones *gather* y índices precalculados, en dos vectores
2. Cargar los *Twiddle Factors*, precalculados, en otro vector
3. Multiplicar uno de los vectores del primer paso por los *Twiddle Factors* y sumarlo al otro vector
4. Guardar los resultados

Esta simplificación no tiene en cuenta que se deben tratar las partes reales e imaginarias por separado o cómo se precomputan los índices de los *gathers*.

Como se puede vislumbrar, con esta implementación se consigue una longitud vectorial igual a N, siendo ésta el número de elementos de la FFT. No obstante, también tiene un posible inconveniente. En cada etapa debemos leer y escribir todos los elementos en memoria para realizar la reordenación.

En la Figura 21 se puede ver la comparación entre la FFTP 1 y la ASLFFTW.

Podemos observar como la FFTP 1 rinde mejor hasta N=1024. A partir de allí, se puede ver en el gráfico de MFLOPS que la FFTP 1 deja de escalar mientras que la de ASLFFTW sigue creciendo hasta N=65536. Mirando los contadores hardware de la Tabla 6 podemos observar como la FFTP 1 utiliza menos instrucciones hasta el tamaño 16384. Si nos fijamos, el número de instrucciones no crece significativamente hasta que pasamos de 256 a 512, momento en el que ya ha utilizado la longitud máxima de vector.

Otro aspecto a destacar de la FFTP 1 es que de las operaciones vectoriales realizadas sobre los elementos vectoriales, el 41 % corresponde a instrucciones de memoria. Por otro lado, la librería ASLFFTW reduce a más de la mitad este valor. El mayor número de accesos a memoria viene acompañado por un *Miss Ratio* mucho mayor para la FFTP 1. Esto acaba repercutiendo en el ratio de ciclos vectoriales dividido por instrucciones vectoriales, que nos aproxima cuánto duran las instrucciones vectoriales. Estos últimos factores parecen ser los responsables del peor rendimiento de la FFTP 1 para tamaños grandes.

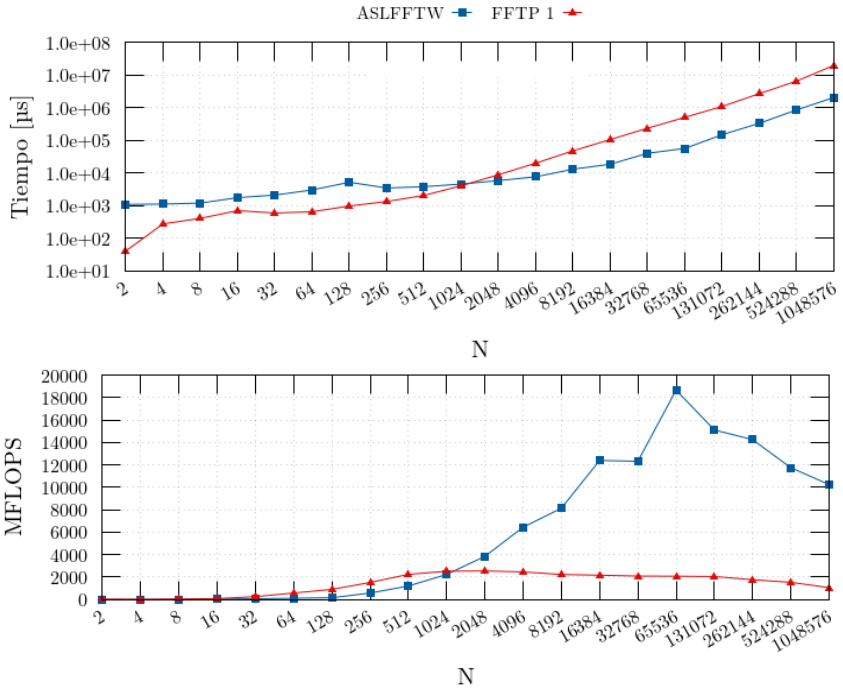

Figura 21: Comparación entre tiempos (arriba) y MFLOPS (abajo) de FFTP 1 y ASLFFTW.

	M. Inst		Longitud Vectorial		Ratio V.Load.E		V.Load MR		Cyc/inst vec	
N	ASLFFTW	FFTP1	ASLFFTW	FFTP1	ASLFFTW	FFTP1	ASLFFTW	FFTP1	ASLFFTW	FFTP1
2	1.05	0.10	1.00	2.00	0.33	0.41	0.00	0.00	10.17	12.68
4	1.08	0.15	1.00	4.00	0.25	0.41	0.00	0.00	4.86	11.13
8	1.17	0.19	1.00	8.00	0.19	0.41	0.00	0.00	3.57	8.84
16	2.34	0.24	1.00	16.00	0.21	0.41	0.00	0.00	4.03	13.36
32	3.38	0.29	1.00	32.00	0.18	0.41	0.00	0.00	2.67	4.17
64	5.22	0.34	1.00	64.00	0.16	0.41	0.00	0.00	2.10	2.88
128	9.19	0.38	1.00	128.00	0.14	0.41	0.00	0.00	1.87	3.62
256	5.18	0.43	16.00	256.00	0.25	0.41	0.00	0.00	3.82	3.51
512	6.21	0.72	22.40	256.00	0.24	0.41	0.00	0.00	3.45	6.04
1024	7.57	1.33	32.00	256.00	0.22	0.41	0.00	0.00	2.85	7.53
2048	9.41	2.65	44.43	256.00	0.21	0.41	0.00	0.00	2.81	7.85
4096	11.90	5.48	64.00	256.00	0.20	0.41	0.00	0.00	2.59	8.33
8192	15.86	11.55	88.62	256.00	0.19	0.41	0.00	0.00	3.31	9.32
16384	21.11	24.53	128.00	256.00	0.18	0.41	0.00	0.00	2.97	9.63
32768	27.57	52.19	176.14	256.00	0.17	0.41	0.00	0.02	3.96	10.02
65536	36.93	110.97	256.00	256.00	0.16	0.41	0.00	0.29	3.99	10.11
131072	82.30	235.40	256.00	256.00	0.17	0.41	0.00	0.41	4.72	10.23
262144	183.18	498.08	256.00	256.00	0.18	0.41	0.02	0.44	4.76	10.53
524288	373.23	1051.06	256.00	256.00	0.17	0.41	0.14	0.68	5.98	11.02
1048576	762.28	2212.29	256.00	256.00	0.17	0.41	0.22	0.81	7.15	16.89

Tabla 6: Comparación entre FFTP 1 y ASLFFTW

6.3. Implementaciones alternativas

Aunque obtener un mejor rendimiento hasta N=1024 es un logro no despreciable, se han ideado otras implementaciones de la FFT intentando solucionar los problemas de la primera versión.

La primera alternativa, que recibe el nombre de FFTP 2, optimiza las operaciones aritméticas que se realizan sobre los datos. La idea que hay detrás es que siendo N el número de muestras, se cumple que $W_N^0 = 1$ y $W_N^{N/2} = -1$. Esto convierte algunas multiplicaciones en sumas y restas. Si revisamos la Figura 20 podemos observar

como en las fases iniciales la mayoría de *Twiddle Factors* se corresponden con estos valores.

Además, la reordenación de los elementos que se deben multiplicar por los *Twiddle Factors* sencillos es más simple y no requiere de gathers. Por contra, la reordenación de los elementos que se deben multiplicar es más compleja. Ahora se requieren tanto *Gathers* como *Scatters*(como el *Gather* pero en escritura) para estos elementos. La implementación FFTP 2 divide cada fase en los elementos que se deben sumar y restar y en los que se deben multiplicar por *Twiddle Factors*.

Otra implementación con la que se ha experimentado es la FFTP 3. Ésta tiene como objetivo reducir enormemente la necesidad de loads y stores. La reordenación que requiere cada fase ahora se realiza en los registros vectoriales y no en memoria con *Gathers*. Como veremos más adelante, realizar la reordenación de esta forma aumenta el número de instrucciones, ya que lo que antes se realizaba con un *Gather* ahora requiere de una serie de desplazamientos.

En la Figura 22 se muestra la comparación entre las tres versiones de la FFTP. Observamos que para la mayoría de tamaños, la primera implementación es la más eficiente. Para algunos casos concretos, la FFTP 2 obtiene un ligero beneficio. Aunque la FFTP 3 prometia un mejor rendimiento al reducir las operaciones de memoria, parece que el impacto de añadir más instrucciones vectoriales ha sido muy relevante.

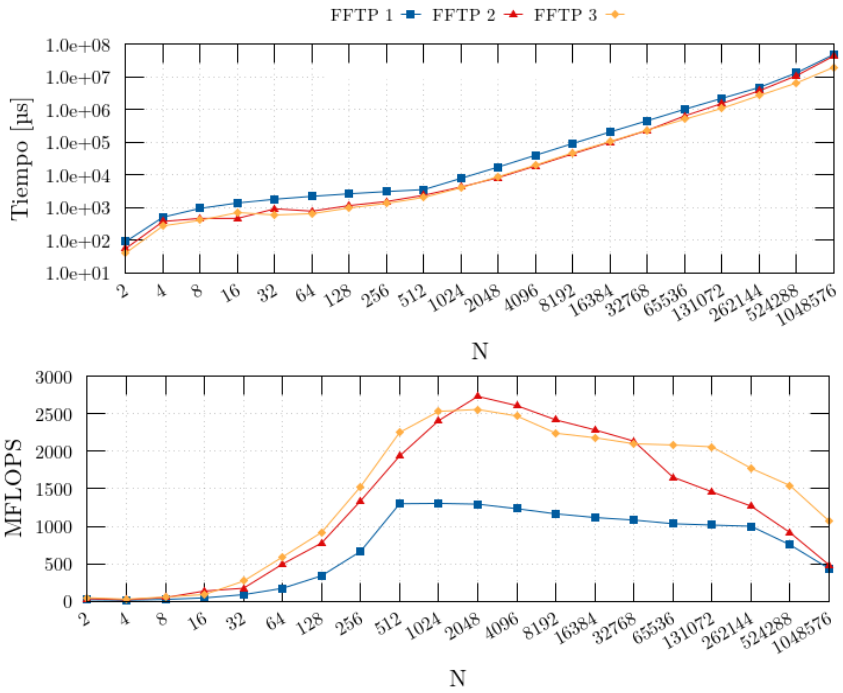

Figura 22: Comparación del tiempo (arriba) y MFLOPS (abajo) de FFTP 1, FFTP 2 y FFTP 3.

Para obtener más información sobre sus diferencias, en la Tabla 7 comparamos los contadores hardware de las tres versiones.

N	M. Inst			Longitud Vectorial			Ratio V.Load.E			V.Load MR			Cyc/inst vec		
	FFTP1	FFTP2	FFTP3	FFTP1	FFTP2	FFTP3	FFTP1	FFTP2	FFTP3	FFTP1	FFTP2	FFTP3	FFTP1	FFTP2	FFTP3
2	0.10	0.16	0.12	2.00	1.37	1.37	0.41	0.38	0.38	0.00	0.00	0.00	12.68	4.12	6.66
4	0.15	0.27	0.19	4.00	1.57	2.38	0.41	0.34	0.27	0.00	0.00	0.00	11.13	9.05	19.26
8	0.19	0.39	0.24	8.00	2.84	4.51	0.41	0.32	0.23	0.00	0.00	0.00	8.84	6.72	24.03
16	0.24	0.50	0.29	16.00	5.52	8.77	0.41	0.30	0.20	0.00	0.00	0.00	13.36	4.78	26.40
32	0.29	0.61	0.34	32.00	10.97	17.23	0.41	0.29	0.18	0.00	0.00	0.00	4.17	7.31	27.56
64	0.34	0.73	0.39	64.00	21.90	34.06	0.41	0.29	0.17	0.00	0.00	0.00	2.88	5.05	28.31
128	0.38	0.84	0.44	128.00	43.82	67.53	0.41	0.28	0.16	0.00	0.00	0.00	3.62	6.25	28.98
256	0.43	0.96	0.50	256.00	87.72	134.18	0.41	0.28	0.16	0.00	0.00	0.00	3.51	7.21	29.70
512	0.72	1.09	0.60	256.00	171.97	256.00	0.41	0.28	0.15	0.00	0.00	0.00	6.04	9.74	29.18
1024	1.33	1.58	1.24	256.00	210.12	256.00	0.41	0.28	0.15	0.00	0.00	0.00	7.53	9.56	29.22
2048	2.65	2.60	2.65	256.00	233.00	256.00	0.41	0.27	0.15	0.00	0.00	0.00	7.85	9.31	29.56
4096	5.48	4.76	5.68	256.00	244.97	256.00	0.41	0.27	0.14	0.00	0.00	0.00	8.33	10.25	31.11
8192	11.55	9.34	12.19	256.00	250.81	256.00	0.41	0.27	0.14	0.00	0.00	0.00	9.32	11.26	33.04
16384	24.53	19.07	26.07	256.00	253.57	256.00	0.41	0.27	0.14	0.00	0.00	0.00	9.63	12.00	34.57
32768	52.19	39.71	55.61	256.00	254.86	256.00	0.41	0.27	0.14	0.02	0.01	0.00	10.02	12.94	35.80
65536	110.97	83.40	118.20	256.00	255.47	256.00	0.41	0.27	0.14	0.29	0.41	0.36	10.11	16.66	37.46
131072	235.40	175.60	250.43	256.00	255.75	256.00	0.41	0.27	0.13	0.41	0.47	0.81	10.23	18.84	38.17
262144	498.08	369.68	528.95	256.00	255.88	256.00	0.41	0.27	0.13	0.44	0.72	0.86	10.53	21.23	38.83
524288	1051.06	777.27	1114.17	256.00	255.94	256.00	0.41	0.27	0.13	0.68	0.96	0.89	11.02	29.96	51.68
1048576	2212.29	1631.32	2340.92	256.00	255.97	256.00	0.41	0.27	0.13	0.81	0.97	0.92	16.89	57.13	90.34

Tabla 7: Comparación de los contadores hardware entre FFTP 1, FFTP 2 y FFTP 3

Comprobamos que las versiones 2 y 3 utilizan vectores más cortos, hecho que impacta negativamente su rendimiento. La FFTP 2 cumple con la intención de eliminar multiplicaciones y reducir el número de instrucciones, y a su vez reduce el ratio de elementos leídos y escritos en memoria respecto los operados.

Por otro lado, la FFTP 3 cumple su objetivo de reducir en gran medida las operaciones de memoria, aunque sufre de una longitud vectorial menor y de un mayor número de instrucciones.

Podemos ver que aunque en las versiones 2 y 3 la presencia de los accesos a memoria sea menor, el ratio de fallos sube hasta números próximos al 100 %. A su vez, los ciclos por instrucción vectorial crecen enormemente.

6.4. Patrones de acceso a memoria

Una vez concluido que la mejor versión por el momento es la FFTP 1 y que uno de sus puntos débiles es la alta presencia de operaciones de acceso a memoria, se ha intentado optimizar esta parte de la implementación.

En primer lugar, se ha eliminado una de las arrays que hacía de buffer. Hasta ahora, habían dos arrays en las que en cada fase se leían y escribían los datos de manera alterna. En la última fase, la array de escritura se convertia en la array de output de la función.

En la versión mejorada se elimina una de estas arrays añadidas. El patrón de las dos arrays sigue estando presente, pero en este caso son siempre la array de salida y una array extra. En función de la paridad del número de fases, se empieza utilizando una o otra para asegurar que en la última fase se escriba en la array de salida.

También se han utilizado las instrucciones de acceso a memoria con más control sobre la cache. Los denominados *Low Priority Loads* permiten que el programador

indique que ciertos loads no se van a reutilizar y, por lo tanto, ponerlos en la cache puede ser contraproducente si expulsan otros bloques que sí que se reutilizarían. Con estas instrucciones no nos aseguramos que estos loads no entren en la cache, ya que es el procesador quien lo decidirá, pero le ayudamos a tomar esta decisión.

Si recordamos la implementación FFTP 1, en cada fase se leían y escribían los N elementos de la transformada y también se cargaban N *Twiddle Factors*. Para cargar los elementos de la transformada también se requería de un load de sus índices, ya que posteriormente se realizaba un *Gather*.

Como tanto los índices como los *Twiddle Factors* estan precomputados para cada fase, nunca se reutilizan. Se han utilizado las instrucciones con baja prioridad para éstos. En teoria esto permite que baje la frecuencia con la que se expulsan los bloques que sí que se reutilizan, los de los valores de entrada y salida de cada fase. El impacto de estas mejoras no son despreciables: en la Tabla 8 se observa como para los tamaños grandes del problema se consigue una mejora de hasta 1.33 en los MFLOPS. Se utiliza "FFTP 1-LP" para indicar la versión con loads de *Low Priority*.

	MFLOPS		
N	FFTP1	FFTP 1- LP	Mejora
65536	2077.95	2083.05	1.002
131072	1915.92	2058.19	1.074
262144	1622.06	1775.51	1.095
524288	1533.88	1565.29	1.020
1048576	800.94	1065.09	1.330

Tabla 8: Impacto en los MFLOPS de los loads de baja prioridad en los tamaños grandes.

Se han medido los contadores hardware que indican el porcentaje de fallos de cache de las operaciones vectoriales. Como se puede ver en la Tabla 23, la versión con loads de baja prioridad reduce ligeramente el porcentaje.

	Miss Ratio		
N	FFTP1	FFTP 1- LP	Mejora
65536	0.420	0.307	1.37
131072	0.458	0.415	1.10
262144	0.582	0.442	1.32
524288	0.792	0.683	1.16
1048576	0.815	0.806	1.01

Figura 23: Impacto en el Miss Ratio de los loads de baja prioridad en los tamaños grandes.

6.5. Versión final

Después de compartir los resultados con otros investigadores que trabajan con la máquina de NEC, se han investigado otras implementaciones diferentes de la FFT.

Todas las FFTP implementadas estaban basadas en la FFT de Cooley-Tukey y sus diferencias residían en la vectorización del algoritmo.

Con la intención de conseguir un rendimiento más próximo o mejor que el de la ASLFFTW, se ha implementado una FFT que sigue el algoritmo de Pease [19]. La primera diferencia de este algoritmo y el ya implementado es que el nuevo sigue el esquema de la *Decimation in Frequency* en vez de *Decimation in Time*. Esto implica que los elementos se reordenan al final de la transformada y no al principio.

Por otro lado, el algoritmo de Pease tiene lo que se denomina *Geometria Constante*[20]. Como se puede ver en la Figura 24, los elementos que se operan entre sí en cada fase son siempre los mismos. Este patrón permitiría no tener que realizar reordenaciones hasta el final, donde tan solo hace falta una reordenación para obtener el resultado correcto. También podemos ver que solo la mitad de los elementos requieren multiplicarse por un Twiddle Factor.

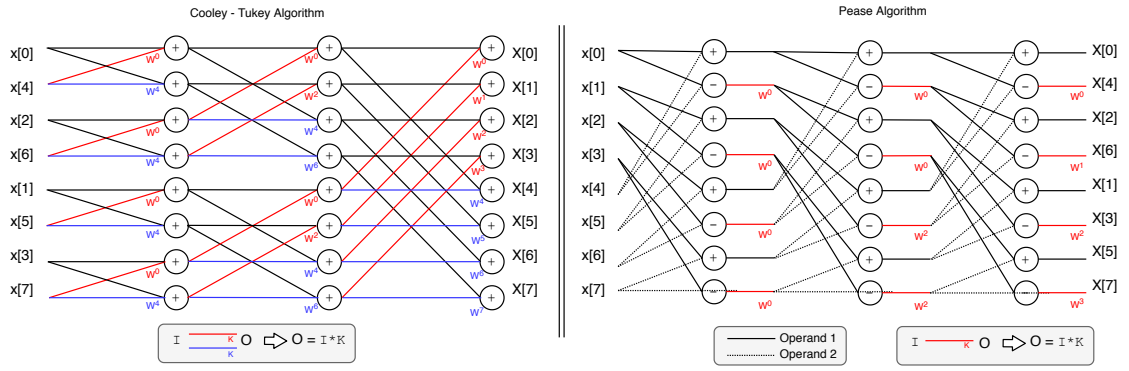

Figura 24: Comparación entre el algoritmo de Cooley-Tukey y Pease para N=8

En este proyecto se ha desarrollado una implementación vectorial de este código. En cada fase, se carga la mitad superior de la array en un vector y la inferior en otro. Se ha hecho esta separación ya que en todas las operaciones de la fase, una mitad es un operando y la otra mitad es el otro. Después se realizan las operaciones de suma y de resta con multiplicación, en dos vectores separados. Al acabar se guardan ambos vectores para la siguiente fase.

Requerimos guardar los vectores y luego volverlos a cargar en la siguiente fase ya que debemos entrelazar los resultados de las sumas y de las restas con multiplicación. En futuras implementaciones se podría experimentar con realizar estos pasos con desplazamientos y no en memoria.

De cualquier forma, esta implementación del algoritmo de Pease no requiere de *Gathers* ni *Scatters* en las fases intermedias, así que aunque siga teniendo que pasar por memoria la localidad espacial es mejor, dado que se acceden a elementos próximos. El único momento en que se realizan *Scatters* es en la reordenación final.

Para mejorar aún más la localidad espacial en memoria de la implementación,

se ha decidio alterar la manera en la que se guardan los números complejos en memoria. Cuando revisábamos las transposiciones en la sección 5.2, se comentó que se guarda la parte real e imaginaria de los elementos de manera entrelazada.

Como los vectores y por lo tanto sus operaciones son de doubles, realizamos tanto las sumas y restas como las multiplicaciones con vectores distintos para la parte real e imaginaria de los complejos. Por lo tanto, en memoria tenemos la partes reales e imaginarias entrelazadas pero luego en los vectores las tenemos separadas.

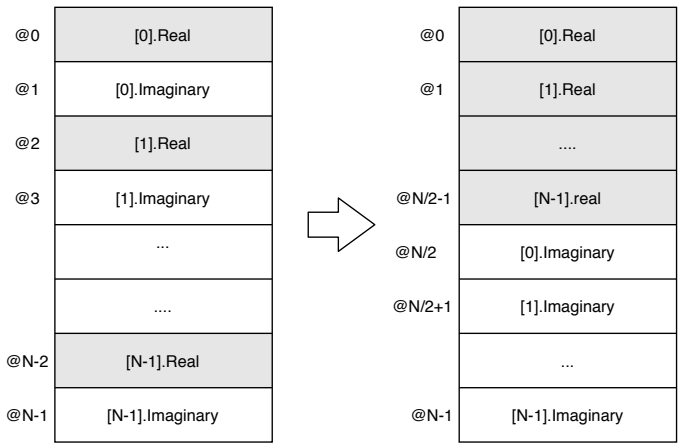

Figura 25: Nuevo esquema de memoria.

En la Figura 25 se muestra gráficamente el nuevo esquema de memoria. Se ha decidido que en las fases intermedias de la FFT se guardarán los componentes reales en la parte superior de la array y los imaginarios en la inferior. Ésto permite leer los elementos de manera contigua, y por lo tanto mejorar la localidad espacial de la cache.

A continuación se observa el rendimiento final de la nueva versión, denominada FFTP 4. En la Figura 26 se muestra el rendimiento de la nueva versión comparada con FFTP 1 y con ASLFFTW.

Observamos que la FFTP 4 rinde mejor que la FFTP 1 tanto en tamaños pequeños como en tamaños grandes. Esto implica que la FFTP 4 sea mejor que la ASLFFTW para tamaños superiores que con la FFTP 1. La FFTP 1 dejaba de superar la ASLFFTW en N=1024, y la FFTP 4 llega hasta N=4096.

En la Tabla 9 se cuantifica la ganancia entre FFTP 4 y FFTP 1 en la última columna, y podemos ver como se consigue una mejora en el tiempo de ejecución de hasta 3.335x para N=1048576.

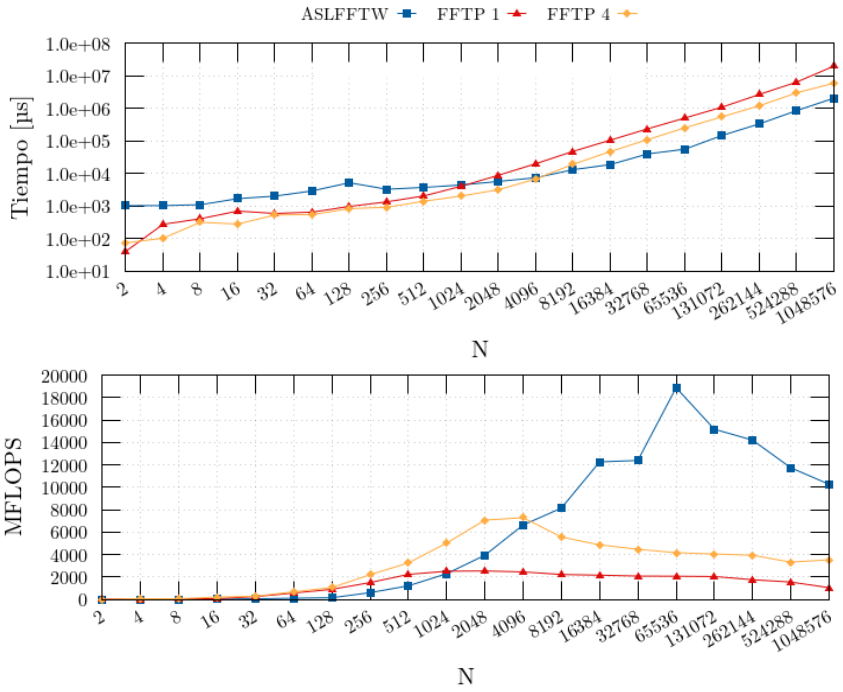

Figura 26: Tiempo (arriba) y MFLOPS (abajo) de la FFTP 4 comparada con ASLFFTW y FFTP 1.

N	tiempo ASLFFTW	tiempo FFTP 1	tiempo FFTP 4	Speedup 4vs1	Speedup 4vsASLFFTW
2	1022.25	41.00	74.00	0.554	13.814
4	1026.00	275.00	103.00	2.670	9.961
8	1097.25	408.75	317.75	1.286	3.453
16	1681.50	701.00	279.75	2.506	6.011
32	2011.00	591.75	532.00	1.112	3.780
64	2909.00	649.75	547.00	1.188	5.318
128	5187.25	975.00	831.25	1.173	6.240
256	3270.75	1340.75	917.25	1.462	3.566
512	3729.00	2047.75	1407.00	1.455	2.650
1024	4425.00	4046.25	2030.00	1.993	2.180
2048	5688.75	8811.50	3178.00	2.773	1.790
4096	7392.25	19909.25	6733.75	2.957	1.098
8192	13060.00	47557.50	19097.25	2.490	0.684
16384	18696.75	105237.25	47014.50	2.238	0.398
32768	39644.00	234101.50	109471.25	2.138	0.362
65536	55551.00	503384.00	250641.75	2.008	0.222
131072	146794.75	1082610.75	547858.00	1.976	0.268
262144	332144.00	2657592.25	1193325.75	2.227	0.278
524288	848021.75	6363959.75	2981779.50	2.134	0.284
1048576	2041957.75	19689823.50	5904088.25	3.335	0.346

Tabla 9: Speedup de FFTP 4 en referencia a FFTP 1 y ASLFFTW.

Por último, compararamos los contadores hardware de las dos versiones de la FFTP y la referencia, ASLFFTW, en la Tabla 10.

	M. Inst			Longitud Vectorial			Ratio V.Load.E			V.Load MR			Cyc/inst vec		
N	ASLFFTW	FFTP1	FFTP4	ASLFFTW	FFTP1	FFTP4	ASLFFTW	FFTP1	FFTP4	ASLFFTW	FFTP1	FFTP4	ASLFFTW	FFTP1	FFTP4
2	1.05	0.101	0.101	1	2	1	0.33	0.41	0.30	0.00	0.00	0.00	10.17	12.68	3.33
4	1.08	0.148	0.153	1	4	2	0.25	0.41	0.32	0.00	0.00	0.00	4.86	11.13	2.86
8	1.17	0.195	0.213	1	8	4	0.19	0.41	0.32	0.00	0.00	0.00	3.57	8.84	7.12
16	2.34	0.242	0.269	1	16	8	0.21	0.41	0.32	0.00	0.00	0.00	4.03	13.36	4.75
32	3.38	0.289	0.323	1	32	16	0.18	0.41	0.33	0.00	0.00	0.00	2.67	4.17	7.58
64	5.22	0.336	0.379	1	64	32	0.16	0.41	0.33	0.00	0.00	0.00	2.10	2.88	6.57
128	9.19	0.383	0.433	1	128	64	0.14	0.41	0.33	0.00	0.00	0.00	1.87	3.62	8.70
256	5.18	0.430	0.489	16	256	128	0.25	0.41	0.33	0.00	0.00	0.00	3.82	3.51	8.46
512	6.21	0.720	0.543	22	256	256	0.24	0.41	0.33	0.00	0.00	0.00	3.45	6.04	11.71
1024	7.57	1.334	1.010	32	256	256	0.22	0.41	0.33	0.00	0.00	0.00	2.85	7.53	7.63
2048	9.41	2.650	2.012	44	256	256	0.21	0.41	0.33	0.00	0.00	0.00	2.81	7.85	5.45
4096	11.90	5.478	4.174	64	256	256	0.20	0.41	0.33	0.00	0.00	0.00	2.59	8.33	5.31
8192	15.86	11.546	8.818	89	256	256	0.19	0.41	0.33	0.00	0.00	0.00	3.31	9.32	6.96
16384	21.11	24.526	18.768	128	256	256	0.18	0.41	0.33	0.00	0.00	0.00	2.97	9.63	7.99
32768	27.57	52.194	39.996	176	256	256	0.17	0.41	0.33	0.00	0.02	0.00	3.96	10.02	8.70
65536	36.93	110.966	85.130	256	256	256	0.16	0.41	0.33	0.00	0.29	0.02	3.99	10.11	9.36
131072	82.30	235.402	180.758	256	256	256	0.17	0.41	0.33	0.00	0.41	0.28	4.72	10.23	9.63
262144	183.18	498.078	382.756	256	256	256	0.18	0.41	0.33	0.02	0.44	0.35	4.76	10.53	9.93
524288	373.23	1051.058	808.240	256	256	256	0.17	0.41	0.33	0.14	0.68	0.55	5.98	11.02	11.71
1048576	762.28	2212.294	1702.206	256	256	256	0.17	0.41	0.33	0.22	0.81	0.95	7.15	16.89	11.06

Tabla 10: Comparación de los hardware counters entre ASLFFTW, FFTP 1 y FFTP 4

Podemos ver que FFTP 4 utiliza un número muy parecido de instrucciones a FFTP 1. A su vez, utiliza la mitad de la longitud vectorial. Si con el mismo número de instrucciones utiliza vectores más pequeños, está realizando menos operaciones que la primera versión.

A su vez, para tamaños grandes también utiliza menos instrucciones que la FFTP 1. Esto se debe a que al utilizar una longitud vectorial menor tarda más en saturar la longitud vectorial máxima y en empezar a cuadriplicar el número de instrucciones.

7. Conclusiones

El panorama de las arquitecturas utilizadas en la computación de alto rendimiento se hace cada día más variado. En esta variedad de sistemas se pueden identificar algunas tendencias comunes: una de ellas es la tendencia emergente de incluir unidades vectoriales que operan sobre registros vectoriales largos que permiten ejecutar una instrucción sobre una gran cantidad de datos.

Tanto Arm con SVE, como RISC-V con la extensión V, así como NEC con el acelerador SX-Aurora son ejemplos de estas tendencias.

Por esta razón, este trabajo se ha centrado en la evaluación de la primera implementación hardware de estas arquitecturas con vectores largos, el procesador NEC SX-Aurora.

El foco de la evaluación se ha fijado en un kernel concreto muy común en muchas aplicaciones paralelas, el cálculo de la transformada de Fourier mediante el método FFT.

Se han explorado dos vías distintas:

1. Modificar la manera en que se llaman las funciones ya existentes con la intención de descubrir combinaciones de llamadas que aprovechasen el cálculo vectorial.

2. Implementar una libería FFT desde zero aprovechando las intrucciones intrínsecas del acelerador NEC SX-Aurora.

Mientras que la primera vía ha proporcionado una mejora de rendimiento muy limitada a un único caso de tamaño específico de la FFT, el segundo camino ha permitido explorar y experimentar más con el acelerador de NEC. Se ha conseguido mejorar el rendimiento en comparación con la librería ASLFFTW con un speedup entre 1.10 y 13.81 para cualquier FFT de hasta 4096 elementos. Se han explorado muchas opciones ofrecidas por la arquitectura para optimizar nuestro código utilizando las instrucciones vectoriales y también se han estudiado diferentes algoritmos FFT, logrando realizar un estudio exhaustivo y completo.

Durante la investigación se han descubierto implementaciones de la FFT potencialmente más eficientes para una máquina vectorial, pero no se han llegado a implementar por falta de tiempo. Se deja este trabajo como continuación de este TFG, provando algoritmos distintos que consigan reducir el tiempo de ejecución limitando las instrucciones de acceso a memoria, ya que estas últimas parecen ser el factor limitante de la implementación propuesta en este TFG.

Referencias

[1] What is high performance computing? https://insidehpc.com/hpc-basic-training/what-is-hpc/.

[2] Josh Stewart. An investigation of simd instruction sets, university of Ballarat School of Information Technology and Mathematical Sciences. http://noisymime.org/blogimages/SIMD.pdf.

[3] NEC SX-Aurora TSUBASA. https://www.nec.com/en/global/solutions/hpc/sx/index.html.

[4] Intel® Architecture Instruction Set Extensions and Future Features... https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-and-future-features-programming-r
html. Library Catalog: software.intel.com.

[5] Arm Ltd. Neon – Arm. https://www.arm.com/why-arm/technologies/neon. Library Catalog: www.arm.com.

[6] N.Stephens. The ARM Scalable Vector Extension. http://arxiv.org/abs/1803.06185, March 2017. arXiv: 1803.06185.

[7] NEC SX-Aurora Intrinsics. https://sx-aurora-dev.github.io/velintrin.html.

[8] Meinard Müller. Fourier Analysis of Signals. http://link.springer.com/10.1007/978-3-319-21945-5_2, 2015.

[9] Ethan. The vocoder and Auto-Tune. https://www.ethanhein.com/wp/2017/the-vocoder-and-auto-tune/, October 2017. Library Catalog: www.ethanhein.com.

[10] Fourier transform infrared (ftir) spectroscopy for analysis of extra virgin olive oil adulterated with palm oil. http://bit.ly/FFTolive.

[11] D.N. Rockmore. The FFT: an algorithm the whole family can use. http://ieeexplore.ieee.org/document/814659/, February 2000.

[12] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. https://ieeexplore-ieee-org.recursos.biblioteca.upc.edu/document/681704, May 1998. ISSN: 1520-6149.

[13] Intel® Math Kernel Library Developer Reference. https://www.intel.com/content/www/us/en/develop/articles/mkl-reference-manual.html. Library Catalog: software.intel.com.

[14] Performance Application Programming Interface. https://icl.utk.edu/papi/.

[15] PROGINF/FTRACE User's Guide. https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf.

[16] RiscV-V. https://github.com/riscv/riscv-v-spec.

[17] The Fast Fourier Transform (FFT) Algorithm (c) - YouTube. https://www.youtube.com/watch?v=lGCAlv3G8Oc&list=LL33Qx83qVgcKTWyAduQ_rWw&index=10&t=1007s.

[18] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. http://www.ams.org/jourcgi/jour-getitem?pii=S0025-5718-1965-0178586-1, May 1965.

[19] Marshall C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. https://doi.org/10.1145/321450.321457, April 1968.

[20] Constant Geometry Parallel Fast Fourier Transform. https://stevescott.ca/2013-05-21-constant-geometry-parallel-fast-fourier-transform.html.